Microarchitectuuronafhankelijke analytische prestatiemodellering
van het spronggedrag en de meerdradige uitvoering van computerprogramma's

Microarchitecture-Independent Analytical Branch Behavior
and Multi-Threaded Performance Modeling

Sander De Pestel

Promotor: prof. dr. ir. L. Eeckhout
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. K. De Bosschere
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2018 - 2019

ISBN 978-94-6355-201-1
NUR 980
Wettelijk depot: D/2019/10.500/9

Examination Committee

Prof. dr. ir. Luc Taerwe, *chair*
Department of Structural Engineering,
Faculty of Engineering and Architecture
Ghent University

Prof. dr. ir. Koen De Bosschere, *secretary*
Department of Electronics and Information Systems,
Faculty of Engineering and Architecture
Ghent University

Prof. dr. ir. Lieven Eeckhout, *supervisor*
Department of Electronics and Information Systems,
Faculty of Engineering and Architecture
Ghent University

Prof. dr. ir. Mario Pickavet
Department of Information Technology,
Faculty of Engineering and Architecture
Ghent University

Prof. dr. ir. Magnus Jahre
Department of Computer Science,
Faculty of Information Technology and Electrical Engineering
Norwegian University of Science and Technology

Dr. ir. Stijn Eyerman
Intel Belgium

Dr. ir. Maximilien Breughe
NVIDIA, Texas, USA

Dankwoord

Dit doctoraat is een werk van lange adem geweest, met periodes waarin alles vanzelf leek te gaan, maar ook periodes waarin ik vooral het gevoel had vast te zitten en geen vooruitgang meer leek te maken. In tijden dat je de wind in de zeilen hebt en alles goed gaat, heb je het misschien niet altijd door. Maar als het dan even tegen gaat, besef je dat er een heleboel mensen zijn die je maar al te graag door deze periode heen helpen, allemaal op hun eigen manier. Die mensen zou ik graag bedanken.

In het bijzonder wil ik mijn promotor Lieven Eeckhout bedanken. Zonder zijn hulp en inzicht zou dit doctoraat nooit tot stand gekomen zijn. Lieven slaagde er altijd in mij uit te dagen om net die stap extra te zetten, die ene cruciale vraag te stellen waar ik eerst even niet goed van was, maar die mijn resultaten uiteindelijk ten goede kwam. Ook Stijn Eyerman, waarmee ik de eerste jaren van mijn doctoraat heel goed heb samengewerkt, wil ik extra bedanken. Door zo aanwezig te zijn en steeds tijd vrij te maken om mij te helpen, heb ik in heel korte tijd heel veel bijgeleerd, niet alleen over computerarchitectuur, maar ook over onderzoek in het algemeen.

I would also like to thank the members to the jury for taking the time to read my thesis, challenge me during the first meeting and provide useful feedback that made the final result better. I would like to address a special word of thanks to Prof Magnus Jahre and Dr Maximilien Breughe who, despite of their busy schedules, accepted my invitation to be a part of my PhD committee and were willing to travel to Ghent.

Familie zijn de mensen die je maken tot wie je bent. De mensen die daar al het langst mee bezig zijn, zijn mijn ouders Freddy en Ann. Zij hadden van in het begin door dat ik een beetje een speciaal geval was en op de juiste manier gemotiveerd moest worden. Ze zijn die uitdaging aangegaan en hebben mij alle kansen gegeven die ik nodig had. Ik hoop dan ook dat zij, net als ikzelf, fier zijn op het resultaat. Daarom wil ik hen, maar ook mijn broer Stijn, mijn zus Emma en mijn pluspapa Erik bedanken voor hun steun en aanmoediging.

Uiteraard wil ik vooral mijn vrouw Jasmijn bedanken. Zonder haar onvoorwaardelijke steun was er van dit doctoraat niets in huis gekomen. Als ik dan al eens vast zat, en de hoop ver te zoeken was, was jij er altijd om te luisteren. Ook al begreep je er meestal geen jota van, toch bleef je luisteren naar mijn uitleg om af te sluiten met:"Je zal wel een oplossing vinden, ik geloof in jou."

Ook onze zoon Siebe was er altijd (uitgeslapen of niet) om mij (dag en nacht) te helpen mijn gedachten te verzetten. Zonder hem zou dit doctoraat niet hetzelfde betekenen. Ook Siebe zijn broer of zus, het volgende lid van ons gezin, zou ik graag al eens bedanken. Zonder hem of haar zou er geen echte deadline geweest zijn en zou dit doctoraat dus misschien letterlijk nooit zijn afgeraakt.

Doctoral research is sitting behind your ultra wide flat screen and focusing on your own research for the next five years. Sounds boring? Yes and a false statement, because you are definitely not alone. If you have a question or you are having trouble with anything, there is always somebody ready to help. That is why I would like to thank everybody I worked with in the previous five years. Thanks Ajeya, Cecilia, Frederick, Jennifer, Josue, Kartik, Kristof, Lu, Mutaz, Sam, Shoaib, Stijn, Trevor, Wenjie, Wim, Xia, it was a pleasure meeting you all and very nice to work in such a multicultural group. Als de hulp dan eerder technisch van aard was, waren ook Michiel, Marnix, Ronny en Vicky altijd bereid te helpen. Ook hen wil ik hartelijk danken voor de tijd die ze namen om mij te helpen het kluwen die de UGent soms kan zijn te ontwarren.

Ook Sam wil ik graag extra bedanken. We zijn op hetzelfde moment ons doctoraat gestart en hebben ons beiden verdiept in het modelleren van microprocessoren. Hierdoor waren er veel raakvlakken tussen onze onderzoekstrajecten, en had ik tevens een reisgenoot voor de ADEPT vergaderingen en conferenties. Ook de vele interessante discussies over microarchitectuur brachten mij de nodige inzichten die de modellen in dit doctoraat hebben gemaakt tot wat ze zijn. Hopelijk kunnen we dit in de toekomst als collega's bij Intel blijven doen. En omdat de boog ook niet altijd gespannen moet staan, wil ik ook graag Bart bedanken, die ten gepaste tijde eens langs kwam met de laatste nieuwtjes en interessante weetjes. Maar die mij ook geleerd heeft dat als je de onderwijsopdrachten met passie probeert te geven, die zoveel leuker worden. Hij leerde me ook dat je niet alleen voldoening kan halen uit het onderzoek, maar ook uit het onderwijs.

Sander De Pestel
Gent, February 4, 2019

Samenvatting

Gedurende de laatste decennia hebben we een spectaculaire evolutie in processor design meegemaakt. Twintig jaar geleden waren computerarchitecten nog processoren aan het bouwen met één enkele rekenkern met 100 miljoen transistoren aan een klokfrequentie rond de 1.5 GHz. De nieuwste processoren daarentegen kunnen klokfrequenties bereiken tot 4.5 GHz in de 'turbo boost' mode. Ook kunnen ze over meer dan 24 rekenkernen beschikken, die bovendien tegelijkertijd nog verschillende draden kunnen uitvoeren.

Architecten hebben de exponentiële groei in het aantal transistoren gebruikt om de prestatie van een rekenkern te verhogen door het toevoegen van allerlei prestatieverhogende technieken zoals speculatie, pijplijning, out-of-order uitvoering, grote caches, prefetching, enz. Tegenwoordig zijn brede out-of-order processoren de standaard en is het verder verbeteren van de prestatie van één enkele rekenkern moeilijk geworden. Daarom gebruiken architecten het toenemend aantal transistoren tegenwoordig voor het verhogen van het aantal rekenkernen per processor. Door het aantal rekenkernen te verhogen is het mogelijk voor de software-architect om werk simultaan uit te voeren en op die manier de uitvoeringstijd van parallelle programma's te verkorten. Technieken zoals herbestemmingsbuffers, bredere pijplijnen, out-of-order verwerking van instructies maken de processor niet alleen sneller, maar ook complexer.

Door deze toenemende complexiteit is de nood aan snelle en nauwkeurige hulpmiddelen om de nieuwe processoren te evalueren heel groot. Het hulpmiddel bij uitstek is tot nog toe altijd simulatie geweest. De nauwkeurigheid van simulatie is heel hoog, waardoor dit de perfecte oplossing lijkt. Maar omdat deze simulatoren het ontwerp gedetailleerd modelleren, zijn ze heel traag. Voor langdurende computerprogramma's leidt dit tot simulatietijden die zo lang duren dat het eigenlijk niet meer nuttig is om te simuleren.

Bij het simuleren van langlopende programma's kunnen computerarchitecten niet langer vertrouwen op de gedetailleerde simulatoren. Daarom maken ze vaker gebruik van een functionele en/of bemonsterde simulatie. Maar zelfs deze simulatietechnieken zijn niet altijd bruikbaar als langdurende programma's of grote systemen gesimuleerd moeten worden.

Technieken zoals analytische modellen zijn een goede manier om gedetailleerde simulatie aan te vullen. Analytische modellen voorspellen de uitvoeringstijd van programma's door middel van wiskundige modellen. Het ge-

bruik van de resultaten uit functionele simulatie door de wiskundige modellen zorgt ervoor dat ze veel sneller zijn ten opzichte van gedetailleerde simulatie. Desondanks kunnen deze modellen een hoge nauwkeurigheid garanderen. Een voorbeeld van een analytisch model is het intervalmodel [20]. Dit model is gebaseerd op de observatie dat de maximale prestatie of de hoeveelheid instructies die de processor per klokcyclus kan verwerken gelijk is aan de breedte van de pijplijn. De instructiestroom zal echter niet altijd perfect verlopen. Om een nauwkeurige schatting te verkrijgen, modelleert het interval model ook onderbrekingen zoals foutief voorspelde sprongen en/of instructie- en data-cache missers.

Het intervalmodel is in staat om op een snelle en nauwkeurige manier de prestatie van een programma te voorspellen. Voor elk ontwerp in de ontwerpsruimte dient een nieuwe functionele simulatie uitgevoerd te worden. Deze terugkerende kost is een probleem bij een grote ontwerpsruimte. Om dit probleem aan te pakken wordt er gebruik gemaakt van micro-architecturaal onafhankelijke profileringstechnieken. Een dergelijk profiel bevat enkel eigenschappen die onafhankelijk zijn van de onderliggende micro-architectuur waardoor een programma slechts één keer geprofileerd moet worden. Vervolgens kan dit profiel gebruikt worden om micro-architectuurafhankelijke inputs te bepalen voor een analytisch model. Gezien het profiel slechts één keer opgesteld dient te worden is dit dus de perfecte oplossing voor het profileren van grote ontwerpsruimtes.

De grote uitdaging is het bouwen van deze micro-architecturaal onafhankelijke profileringstechnieken. Om het geheugengedrag te modelleren is reeds een model beschikbaar, StatStack genaamd [18]. StatStack maakt gebruik van hergebruiksafstanden tussen geheugentoegangen om te voorspellen wat de prestatie is voor een cache-geheugen van eender welke grootte. Er bestaat ook werk rond het modelleren van spronggedrag zoals 'taken rate' en 'transition rate' [9, 24]. Ook bestaat er een werk van Yokota et al. dat gebruik maakt van sprongentropie [61]. Bij het gebruik van deze technieken kan er echter geen nauwkeurige voorspelling gemaakt worden van de kost ten gevolge van foutief voorspelde sprongen. Daarom is er nood aan een nieuwe techniek.

Wij stellen **lineaire sprongentropie** voor, een nieuwe techniek die bepaalt hoe voorspelbaar het spronggedrag is van een programma. Als de entropie 0 is, betekent dit dat het patroon heel regelmatig is en dat de sprongen makkelijk voorspelbaar zijn. Dit zal leiden tot een laag aantal foutief voorspelde sprongen. Als de entropie 1 is, betekent dit dat er veel willekeurig spronggedrag aanwezig is. Hierdoor zijn sprongen heel moeilijk te voorspellen, wat zal leiden tot een groot aantal foutief voorspelde spongen. In onze techniek is de definitie van entropie overgenomen van Shannon entropie, maar is de binaire formule omgevormd van een som van logaritmen naar een simpele lineaire functie. Onze nieuwe lineaire functie komt beter overeen met de werking van sprongvoorspellers, wat een meer nauwkeurige modellering oplevert.

Omdat verschillende sprongvoorspellers een andere prestatie zullen leveren voor deze programma's, zal elke sprongvoorspeller over zijn eigen model moeten beschikken. We voorspellen het aantal foutief voorspelde sprongen als een

lineaire functie van de sprongentropie. De volgende vergelijking geeft het aantal foutieve voorspellingen M aan in functie van de entropie E:

$$M(E) = \alpha + \beta \times E. \tag{1}$$

In deze formule zullen α en β berekend worden door middel van training. Deze parameters zullen anders zijn voor elke sprongvoorspeller. Wij valideren deze nieuwe techniek door het voorspellen van de voorspellingsnauwkeurigheid van vijf sprongvoorspellers: GAg, GAp, PAp, gshare en een gecombineerde voorspeller, voor 95 programma's uit SPEC CPU 2006 en CPB 2011. Deze techniek levert een gemiddelde fout op van 0.70 MPKI[1] voor CBP en 0.89 MPKI voor de SPEC programma's.

Lineaire sprongentropie kan gebruikt worden voor het sturen van **if-conversie**. Lineaire sprongentropie is een techniek die met een hoge nauwkeurigheid sprongen kan classificeren in gemakkelijk en moeilijk te voorspellen sprongen. Dit gebeurt voor elke individuele sprong in een programma waardoor dit profiel gebruikt kan worden voor het sturen van if-conversie. De invloed van het spronggedrag in een programma hangt af van de hoeveelheid foutief voorspelde sprongen. Met andere woorden, als sprongen heel moeilijk te voorspellen zijn, zal de invloed heel groot zijn. Die impact verkleinen zal moeten gebeuren door het vermijden van deze sprongen. Door onze techniek te implementeren in de LLVM compiler, wordt de prestatie van programma's tot 2.4% verbetert ten opzichte van standaard if-conversie.

Lineaire sprongentropie is ook bruikbaar om de **invloed van spronggedrag** op de prestatie van een programma te bepalen. Eerst wordt een entropieprofiel opgesteld van het programma door het uit te voeren met onze profileringstool. Vervolgens wordt gebruik gemaakt van deze entropie samen met het model van de gebruikte sprongvoorspeller om het aantal foutief voorspelde sprongen te bepalen. Hieruit kan de totale invloed van alle sprongen bepaald worden op de uitvoeringstijd van het programma. Lineaire sprongentropie is gebruikt in het model van Van den Steen et al. [56] dat werd voorgesteld op de ISPASS 2015 conferentie. In dit model is de fout van de sprongcomponent slechts 1% in vergelijking met simulatie. Dit leidt tot een fout van 0,16% op de volledige uitvoeringstijd. Het oude model van Yokota daarentegen onderschat de invloed met gemiddeld 60%. Dit leidt tot een fout van 7% op de totale uitvoeringstijd.

Het model voorgesteld door Van den Steen et al. [56] kan gebruikt worden om de prestatie te schatten van een computerprogramma dat uit slechts één draad bestaat. Maar aangezien softwareprogrammeurs niet langer kunnen rekenen op de computerarchitect om de prestatie van één enkele rekenkern te verbeteren, moeten programma's aangepast worden om hun rekenwerk parallel uit te voeren. Alleen door veel parallel werk uit te voeren kan de software de hardware ten volle benutten. Dit voegt weer extra complexiteit toe, zowel voor de softwareontwikkelaar, als voor de hardwaredesigner. Om de softwareon-

[1] MPKI: Het aantal fouten per 1000 instructies.

twikkelaar te ondersteunen bij het optimaliseren van zijn programma's en om de hardwaredesigner toe te laten snel en efficiënt een grote ontwerpsruimte te analyseren is er een hoge nood aan betere tools die de noodzakelijke inzichten geven.

Om deze nieuwe uitdagingen aan te pakken, stellen we **RPPM** voor, een micro-architecturaal onafhankelijke modelleringstool voor het maken van een snelle prestatieschatting voor meerdradige programma's op parallelle hardware. Meerdradige programma's maken gebruik van meerdere, zogenaamde, werkdraden die het rekenwerk verdelen en parallel kunnen uitvoeren. Door het verdelen van het werk zal het programma hetzelfde werk sneller kunnen uitvoeren, maar hierdoor zal de complexiteit van het programma toenemen. Deze draden zullen elkaar beïnvloeden, zowel direct door synchronisatie als indirect via het geheugen.

Het eerste grote verschil tussen enkeldradige en meerdradige programma's is de directe invloed die de verschillende draden kunnen uitoefenen op elkaar door synchronisatie. Deze functies zijn beschikbaar via verschillende bibliotheken zoals OpenMP, pthread, enz. Draden beïnvloeden elkaar via het geheugen. Om de invloed die draden op elkaar uitoefenen via het geheugen te modelleren, gebruiken we een nieuwe versie van StatStack [1]. Deze tool is in staat om het geheugengedrag van deze meerdradige programma's te modelleren.

Om synchronisatiegedrag te modelleren worden alle oproepen naar de synchronisatiebibliotheken opgevangen tijdens een profileringsfase, dit samen met alle andere eigenschappen die nodig zijn om de prestatie van elke individuele draad te bepalen alsook het geheugengedrag. Om de uitvoeringstijd van meerdradige programma's te bepalen, wordt eerst de uitvoeringstijd van alle draden tussen de verschillende synchronisatiepunten bepaald. Dit gebeurt door het enkeldradige model uit te breiden met de meerdradige StatStack versie. De laatste stap is het terug toevoegen en modelleren van alle synchronisatiegebeurtenissen.

RPPM is in staat de uitvoeringstijd van alle Parsec en Rodinia programma's te voorspellen met een gemiddelde absolute fout van 11%, een aanzienlijke verbetering ten opzichte van naïve uitbreidingen van het enkeldradige model die leiden tot een gemiddelde fout van 28%. Wij tonen ook aan dat RPPM een nuttig model is om op een snelle en nauwkeurige manier een ontwerpsruimte te onderzoeken en inzicht te verkrijgen in het gedrag van een computerapplicatie op toekomstige hardware.

Summary

Over the last few decades, we witnessed a spectacular evolution in processor design. Whereas 20 years ago, architects were building single-core processors with 100 million transistors at clock frequencies around 1.5 GHz, contemporary state-of-the-art processors can reach clock frequencies exceeding 4.5 GHz in a so-called turbo boost mode, and feature more than 24 cores, with multiple threads running simultaneously on the same physical core.

Architects used the exponentially increasing number of transistors through Moore's Law to speed up single-threaded performance through various performance enhancements, including speculation, out-of-order execution, larger caches, etc. Nowadays, superscalar out-of-order processing is the defacto standard and improving single-threaded performance is becoming harder. Therefore, architects are using the available transistor count for scaling up the number of cores in a processor. By increasing the core count, the software architect is able to process data in parallel, thereby significantly increasing performance. But speculation, deeper and wider pipelines, out-of-order processing of instructions, etc. not only make processors faster, they also make them more complex.

Due to the increased complexity, the need for fast and accurate tools to evaluate new processor designs is very high. The preferred tool, until now, for processor design has been detailed simulation. The level of accuracy for cycle-accurate simulation is very high, hence it may appear to be the perfect tool for taking important design decisions. However, since these simulators implement the design in great detail, they are also very time-consuming. This leads to impractical slowdowns when simulating long-running applications.

To simulate long-running applications, architects can no longer rely on detailed simulators and hence need to use functional and/or sampled simulation. Unfortunately, even these simulation techniques become unfeasible when systems become too large or applications too long-running.

Techniques such as analytical models are a good way to complement detailed timing simulations. These models predict the execution time using a mathematical model that uses inputs derived from a simple functional simulation. It does this while maintaining high prediction accuracy. The interval model [20] is one example that is based on the observation that, without miss events, the performance or instructions per cycle (IPC) of the processor pipeline should be equal to the width of the pipeline. This means that the processor is able to pro-

cess instructions at its maximum IPC, e.g., at the designed dispatch width, as long as there are no miss events. This behavior is best observed at the dispatch stage, where instructions leave the front-end of the pipeline (after fetching and decoding is done) and enter the back-end, where the instructions are executed and all communication to memory is handled. The interval model models the impact of branch mispredictions, instruction- and data-cache misses.

The interval model is able to provide a fast and accurate way for predicting application performance, but with a growing design space, even the recurring cost of re-running the functional simulations with different configurations becomes a major concern. Our approach to this problem is the use of a microarchitecture independent profiling technique, such that this profile only consists of application characteristics that are independent of the microarchitecture the application is running on. After building the profile, microarchitecture-specific inputs are generated based on a given hardware configuration, which enables the analytical model to predict performance. Constructing this profile is a one-time cost, thus the perfect solution for the exploration of large processor design spaces.

The key challenge is thus to create microarchitecture-independent profiling techniques to provide inputs to the interval model. To model cache behavior, a tool called StatStack [18] can be used, which uses reuse distance histograms to predict the cache miss rate of an application for any cache size. Some prior work has been done to model branch behavior of an application, e.g., taken rate and transition rate [9, 24], and branch entropy by Yokota et al. [61]. Unfortunately, this prior work is not able to provide an accurate prediction of the number of mispredicted branches for a specific branch predictor. Therefore, a new, more accurate model is needed.

We propose **linear branch entropy**, a novel metric that quantifies how regular the branch behavior of an application is. An entropy of 0 means that there is a regular pattern in the branch behavior. Therefore, these branches are easy-to-predict, leading to a low number of mispredicted branches. When entropy equals 1, this implies that there is a lot of randomness in the branch behavior, therefore these branches are hard-to-predict, leading to a high number of mispredicted branches. We keep this the same as the definition of Shannon entropy, but we adapt the entropy formula from a sum of logarithmic functions to a simple linear function, and we demonstrate that our new approach correlates better with how a branch predictor actually works.

Because different branch predictors have a different misprediction rate for the same application, each type of branch predictor has its own model. We find that a linear relationship between entropy and the measured branch misprediction rate is the best fit. We use the following equation to calculate the misprediction rate M for a given entropy E:

$$M(E) = \alpha + \beta \times E. \tag{2}$$

Parameters α and β are calculated during a training run and are different for every branch predictor. We validate this new metric by predicting the misprediction rate for five branch predictors, e.g., GAg, GAp, PAp, gshare and a tournament predictor, for 95 benchmarks from two benchmark suites, SPEC CPU 2006 and CPB 2011. Across all modeled predictors, the average absolute error is around 0.70 MPKI and 0.89 MPKI for CBP and SPEC, respectively. The average MPKI equals 10.8, which means that the model features a relative error of less than 10%.

The first use case of linear branch entropy is the ability to steer **if-conversion**. Linear branch entropy is an accurate metric for categorizing branches into hard-to-predict and easy-to-predict branches. This can be done on a per-branch basis and is thus an effective tool to profile branches to steer if-conversion. The impact of branch behavior on the overall execution time of an application is influenced by the number of mispredicted branches. When branches are very hard to predict, the branch predictor is not able to avoid most mispredictions. Thus, decreasing the branch impact should be done by avoiding this penalty, through if-conversion. By implementing this technique in the LLVM compiler, we improve performance by up to 2.4% compared to the default implementation of if-conversion.

The second use case of linear branch entropy is the ability to provide inputs to the interval model to model the **branch penalty**. During the profiling run of the application, linear branch entropy is calculated. During the modeling phase, the model of the used branch predictor along with the entropy provides a prediction for the misprediction rate. From this, we can derive the number of mispredicted branches, which in turn, can be used to calculate the impact of all mispredicted branches. The use of linear branch entropy to model the branch penalty is used as part of the single-threaded model proposed by Van den Steen et al. [56]. In this model, the error of the branch component is only 1% compared to simulation, leading to a 0.16% error on the complete execution time, whereas Yokota's model, on average, leads to an underprediction of around 60%, or to a 7% error when predicting overall execution time.

This microarchitecture-independent analyical model can predict the execution time of single-threaded applications, but in order to make the most out of current multicore processor hardware, applications need to adapt. Programmers can no longer rely on the computer architect to improve performance. Therefore, applications need to adapt and parallelize the work to utilize the available hardware to the best possible extent. This adds yet another dimension and even more complexity to the design of both the hardware and the software. To enable the software architect to adapt their applications to upcoming multicore hardware, and to enable the hardware architect to optimize this new hardware, they need new tools that enable them to do a fast and accurate design space exploration, and that give them the necessary insight into the possible performance bottlenecks.

To tackle these new challenges, we propose **RPPM**, a micro-architectural independent modeling tool for rapid performance prediction of multi-threaded applications on multicore hardware. Multi-threaded applications create multi-

ple worker threads that are able to execute in parallel. By dividing the work among these worker threads, the application tries to speed up the execution. But this comes with a cost and significant added complexity. These threads can and will interfere with each other, both directly through synchronization and indirectly through memory.

The first major difference between single- and multi-threaded applications is the interference threads have upon each other through synchronization. The most used synchronization primitives include critical sections, barriers and producer-consumer relationships. These functions are available through parallel execution libraries, such as OpenMP, pthread, etc. Interference through memory can occur in many ways and can both lead to an increase or a decrease of the execution time. To model interference through memory, RPPM uses a new version of StatStack [1], that is able to predict cache miss rates for multi-threaded applications.

To model multi-threaded execution behavior, we catch all calls to the synchronization libraries during the profiling phase, together with all characteristics needed to model per-thread behavior and multi-threaded memory behavior. To predict the execution time of multi-threaded applications, we first predict the execution time of all individual threads, between synchronization events. This is done by extending the single-threaded model with multi-threaded StatStack. The next step is to predict the overhead caused by synchronization, which is done by reintroducing all synchronization events and by modeling their execution behavior.

RPPM predicts the execution time of all Parsec and Rodinia applications with an average absolute error of 11%. This significantly outperforms naive extensions of the single-threaded model that do not model any synchronization, leading to an average error of 28%. We also show that RPPM is a valuable tool to speed up design space exploration and to gain insight into the execution behavior of multi-threaded applications on future designs.

Contents

Examination Committee i

Dankwoord iii

Samenvatting v

Summary ix

List of Figures xvii

List of Tables xxi

List of Abbreviations xxiii

1 Introduction 1

1.1 Motivation 1

1.2 Key Challenges 2

1.3 Thesis Contributions 3

1.4 Other Research Activities 5

1.5 Thesis Overview 6

2 Background 9

2.1 Processor Architecture 9

2.1.1 In-order Processor 9

2.1.2 Caches 11

2.1.3 Out-of-Order Processing 12

2.1.4 Superscalar Out-of-Order Processing 13

2.2 Performance Modeling 14

2.2.1 Simulation 15
2.2.2 Analytical Modeling 17
2.3 Interval Model 18
2.4 Micro-architectural Independent Modeling 22

I Linear Branch Entropy 25

3 Linear Branch Entropy 27
3.1 Introduction 27
3.2 Background 29
3.2.1 Branch Prediction 29
3.2.2 Taken Rate 31
3.2.3 Transition Rate 31
3.2.4 Branch Entropy 31
3.2.5 Branch Predictor as a Markov Predictor 32
3.2.6 Micro-Architecture Independent Characterization 32
3.3 Linear Branch Entropy 32
3.4 Branch Predictor Model 34
3.4.1 Overview 35
3.4.2 Recording Branch Behavior 36
3.4.3 Calculating Branch Entropy 37
3.4.4 Model Construction 38
3.5 Results 39
3.5.1 Experimental Setup 40
3.5.2 Model Accuracy 41
3.5.3 Interpreting the Model Parameters 46
3.5.4 Comparison to Prior Work 47
3.5.5 Modeling Aliasing 48

4 Branch Behavior Analysis 51
4.1 Comparing State-of-the-Art Predictors 51
4.2 2016 Championship Branch Prediction 53
4.3 Representative Benchmark Selection 54
4.4 Impact of History Length 55

5 Modeling Branch Penalty 59
5.1 Branch Misprediction Penalty 59
5.1.1 Predicting the Number of Mispredicted Branches 59
5.1.2 Branch Resolution Time 60
5.2 Results 61

6 Guiding If-Conversion 63
6.1 Prior Work 64
6.2 If-Conversion on x86 using LLVM 64
6.3 Experimental Setup 65
6.4 Results 66
6.4.1 Analysis of bzip2 67
6.4.2 Analysis of gobmk 69
6.4.3 Branch Entropy versus Taken Rate 69
6.4.4 Impact across Microarchitectures 70

II Modeling Multi-Threaded Performance 73

7 Modeling Multi-Threaded Performance 75
7.1 Introduction 75
7.2 Background 77
7.2.1 Accumulating Errors 77
7.2.2 Single-Threaded Performance Model 78
7.2.3 Naive Extensions for Multi-threaded Applications . . . 80
7.3 RPPM 80
7.3.1 RPPM Workflow 81
7.3.2 Memory Interference 82
7.3.3 Synchronization 86
7.4 Experimental Methodology 91
7.4.1 Benchmarks 92
7.4.2 Simulator 94
7.4.3 Profiling 94
7.5 Evaluation 94

8 Behavior of Multi-Threaded Applications 97

8.1 Design Space Exploration 97

8.2 Bottle Graphs 99

8.3 Time Sharing 103

8.4 Parallel Execution Models 106

9 Conclusions and Future Work 109

9.1 Linear Branch Entropy 109

9.2 Multi-Threaded Performance Modeling 110

9.3 Future Work 111

9.3.1 Power Efficiency 111

9.3.2 Impact of Time Sharing and SMT 111

9.3.3 Message Passing Interface (MPI) 112

9.3.4 Heterogeneous Multi-threading 112

Bibliography 113

List of Figures

2.1 Pipeline model of an in-order processor. 10

2.2 Model of a cache hierarchy where two cache levels were added in-between the core and main memory. 12

2.3 Pipeline model of a superscalar out-of-order processor with a wider front-end pipeline and bigger execute stage. 14

2.4 IPC shows an on/off behavior when focusing on the dispatch stage. 17

2.5 Detailed view of the interval timing of a branch misprediction. 18

2.6 Detailed view of the interval timing of an I-cache miss. 19

2.7 Detailed view of the interval timing of a long latency data cache miss. 20

2.8 Schematic overview of how micro-architectural independent modeling works. 22

3.1 A state diagram of a two-bit saturating counter. 29

3.2 Two examples of two-level branch predictors. 30

3.3 Comparison between Shannon's binary entropy and the linear branch entropy we propose. 34

3.4 Schematic overview of how the branch predictor model works. . 35

3.5 The profiler records branch outcomes and history of every unique branch, both for the global and local history. 36

3.6 Reducing the size of the OFT to calculating branch entropy for a smaller history size. 37

3.7 Graphical representation of the training input and the fitted model. 40

3.8 Prediction error as a boxplot, showing median and average absolute error. 42

3.9 Prediction error in MPKI for the different predictors (CBP). . 44

3.9 Prediction error in MPKI for the different predictors (SPEC). . 45

3.10 Average absolute error for the two models on the two benchmark suites for common predictors. 47
3.11 Comparison of the model error to prior work. 48
3.12 Comparison of the model error for different predictor sizes. . . 49
3.13 GAp model with and without taking aliasing into account. . . . 50

4.1 Models for the state-of-the-art predictors of the 2014 CBP championship. 53
4.2 The entropy for all benchmarks used in the 2016 CBP. 54
4.3 Result of PCA analysis and clustering for the 40 benchmarks. . 55
4.4 Impact of the number of history bits on medium and hard to predict branches. 57

5.1 Schematic overview of how the branch predictor model works. . 60
5.2 Branch component as predicted by simulation, linear branch entropy and Yokota et al. 62

6.1 Performance improvement of our new if-conversion technique compared to the standard technique implemented in LLVM. . . 66
6.2 Analysis of the impact of every single convertable branch on the execution time for bzip2. 67
6.3 Performance improvement for gobmk using reference input versus the training input. 69
6.4 Performance improvement for no if-conversion, using only entropy, only taken rate, and both. 70
6.5 Performance improvement on an in-order Intel Atom processor. 71

7.1 Schematic overview of the RPPM tool. 80
7.2 RPPM predicts multi-threaded execution time in three steps. . 82
7.3 Modeling private caches is done in three steps. 83
7.4 Reuse distance can change significantly when moving to a shared cache. 84
7.5 Modeling shared caches involves five steps. 85
7.6 Prediction error for MAIN, CRIT and RPPM. 95
7.7 CPI stacks for RPPM and simulation normalized to simulation. 96

8.1 Bottle graphs for all Parsec benchmarks (part 1). 100
8.1 Bottle graphs for all Parsec benchmarks (part 2). 101

8.2 Bottle graphs obtained through RPPM and simulation for the dedup benchmark. 104

8.3 Bottle graphs obtained through RPPM and simulation for the ferret benchmark. 105

8.4 Bottle graphs for the OpenMP and pthread implementations of the streamcluster benchmark. 107

List of Tables

3.1 Model parameters for the different branch predictors. 46

4.1 Model parameters for the state-of-the-art branch predictors. . . 52
4.2 Entropy and simulated misprediction rate for the top-4 CBP predictors. 52
4.3 Average misprediction rates for entropy- and taken/transition rate based clustering. 56

6.1 Evaluated benchmarks and inputs, along with the number of convertable and converted branches. 65
6.2 Profile and hardware performance counter information for interesting branches in bzip2. 68

7.1 Accumulating prediction errors in barrier-synchronized applications. 78
7.2 Overview of all Rodinia benchmarks used. 92
7.3 Overview of all Parsec benchmarks used. 93
7.4 Simulated architecture configurations. 94

8.1 Case study: Predicting the optimum design point. 98
8.2 Impact of time sharing on dedup and ferret. 103

List of Abbreviations

ABP Average Branch Path

ALU Arithmetic Logic Unit

BHT Branch History Table

CBP Championship of Branch Prediction

CPI Cycles Per Instruction

CPU Central Processing Unit

DRAM Dynamic Random Access Memory

EX Execution

GHz Gigahertz

ID Instruction Decode

IF Instruction Fetch

ILP Instruction-Level Parallelism

IPC Instructions Per Cycle

ISA Instruction-Set Architecture

L1D Level-1 Data Cache

L1I Level-1 Instruction Cache

L2 Level-2 Cache

L3 Level-3 Cache

LLC Last-Level Cache

LRU Least Recently Used

MIPS Million Instructions Per Second

MLP Memory-Level Parallelism

MPKI Misses Per Kilo Instructions

OF Operand Fetch

OoO Out-of-Order

PC Program Counter

PHT Pattern History Table

RD Reuse Distance

ROB Reorder Buffer

ROI Region Of Interest

RPPM Rapid Performance Prediction for Multi-threaded applications

SD Stack Distance

SMT Simultaneous Multithreading

SOTA State-of-the-art

WB Write-back

Chapter 1

Introduction

This chapter introduces the reader to the work that is described in this thesis and ends with an overview of how the book is structured.

1.1 Motivation

In the last decades we witnessed a spectacular evolution in processor design. While 20 years ago architects were still building single-core processors with 100 million transistors at clock frequencies around 1.5 GHz[1], contemporary state-of-the-art processors can reach clock frequencies exceeding 4.5 GHz in so-called turbo boost mode[2], while featuring more than 24 cores, with multiple threads running simultaneously on the same physical core. This is made possible thanks to Moore's law which states that the number of transistors doubles every generation leading to a whopping 10 billion transistors that architects currently have available.

Architects used this increasing number of transistors to speed up single-threaded performance through various enhancements, including pipelining, out-of-order executing, speculation, larger caches, pre-fetching, etc. Nowadays, superscalar out-of-order processing is the norm and improving single-threaded performance is becoming harder, therefore architects are using the available transistor count for scaling up the number of cores in a processor. By increasing core count, the software architect is able to process data in parallel, thereby significantly improving performance. But reorder buffers, wider pipelines, out-of-order processing of instructions, do not only make processors faster, they also make them more complex.

Due to this increased complexity, the need for fast and accurate tools to evaluate new processor designs is very high. The preferred tool, until now, for processor design has been the use of detailed cycle-level simulation. But due to

[1] https://ark.intel.com/products/series/78132/Legacy-Intel-Pentium-Processor
[2] https://ark.intel.com/products/series/122593/8th-Generation-Intel-Core-i7-Processors

increased complexity, simulation time has increased dramatically. This makes detailed simulation impractical for designing future designs. Techniques such as analytical models are a good way to complement detailed timing simulations. These models predict the execution time using a mathematical model that uses inputs derived from a simple functional simulation. It does this while maintaining high prediction accuracy. But with a growing design space, even the recurring cost of re-running the functional simulations with different configurations becomes a major issue.

Our approach to this problem is the use of a micro-architectural independent profiling technique, such that this profile only consists of application characteristics that are independent of the microarchitecture the application is running on. After building the profile, microarchitecture-specific inputs are generated based on a given hardware configuration, needed for the analytical model to make a prediction. Constructing the profile is a one-time cost, making the analytical model a compelling solution for driving big design space explorations.

1.2 Key Challenges

Creating micro-architectural independent profiles means we need to identify characteristics that are not influenced by the hardware where the profiling is running on, and that are measurable without knowing the exact microarchitecture of the target design. Prior work has been done to analyze branch behavior [9, 12, 24, 61], but none of the existing work provides the accuracy needed for our model. Therefore the first challenge was the development of a metric that is both easy to use and delivers the accuracy needed for the model to accurately predict the penalty caused by incorrectly predicted branches.

This new metric gives us valuable insight into the impact branch prediction has on the overall execution time of an application. Now that we have a method to categorize branches based on their behavior, the challenge is to use this information to speed up the execution of an application by minimizing the impact hard-to-predict branches have on performance.

Today, hardware uses more and more parallel processors to increase the total throughput of the chip, but in order to evaluate changes to this new design, simulation is becoming too slow. The main challenge is to provide the hardware architect with tools to quickly and efficiently evaluate the new design choices. These tools are also very important for the software engineer, since developing and optimizing parallel applications is not a trivial problem. Software engineers also need tools to analyze the behavior of an application on new parallel hardware.

1.3 Thesis Contributions

Contribution #1: Linear Branch Entropy.

The first main contribution of this thesis is the use of a technique based on Shannon's entropy theory to model branch behavior in a micro-architectural independent way. The predictability of a branch highly depends on the randomness of the taken/not-taken pattern of that branch. Shannon's formula can be used to translate this randomness into a single number, called entropy. The basic formula however was found to be too complex for practical use.

We therefore simplify Shannon's formula and propose linear branch entropy. This new entropy metric is still a number between 0 and 1, where 0 means easy-to-predict and 1 means hard-to-predict, but now there is a linear relationship between randomness and entropy. This makes it a lot easier to work with, especially when inverting the formula in order to calculate the misprediction rate for a branch predictor based on the entropy of an application. This work resulted in a publication at the 2015 ISPASS conference:

> S. De Pestel, S. Eyerman, and L. Eeckhout. Micro-architecture independent branch prediction modeling. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS), pages 135 -144, March 2015.

Upon publication of the above work, the linear branch entropy technique developed in this thesis was used to validate the coverage of the benchmarks in the 2016 Championship of Branch Prediction or CBP. More precisely, my entropy work was used to verify whether the set of benchmarks contains a sufficiently diverse set of benchmarks from both ends of the spectrum, i.e., containing benchmarks with straightforward branch behavior and hard-to-predict behavior.

> M. Breughe. Maximizing Branch Behavior Coverage for a Limited Simulation Budget. Championship Branch Prediction (CBP-5), June 2016, www.jilp.org/cbp2016/program.html.

Contribution #2: Modeling Branch Misprediction Penalty.

The novel entropy technique developed in this thesis can be used to predict the penalty created by the branch predictor during the execution of an application. Linear branch entropy was integrated in a fellow PhD student's profiling and modeling tool to model the execution time of single-threaded applications. This resulted in two co-authored publications:

> S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T.E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Micro-architecture independent analytical processor performance and power modeling. In Proceedings of the IEEE International Symposium on

Performance Analysis of Systems Software (ISPASS), pages 32-41, March 2015.

S. Van den Steen, S. Eyerman, S. De Pestel, M. Mechri, T.E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Analytical processor performance and power modeling using micro-architecture independent characteristics. IEEE Transactions on Computers (TC), 65(12) pages 3537-3551, 2016.

Contribution #3: Steering If-Conversion.

Entropy is an accurate characteristic to categorize branches into hard-to-predict versus easy-to-predict branches. This can be done on a per-branch basis in a microarchitactural-independent way. It thus is an effective tool to profile branches to steer if-conversion, which is a technique to avoid the penalty caused by incorrect branch predictions. Instead of executing only one path, based on the result from the branch predictor, if-conversion will execute both possible directions. After executing both paths, based on the condition of the branch, it will keep the result from the correct execution path.

Of course, since if-conversion will execute both paths, this is only a good idea for hard-to-predict branches, where the benefits of avoiding the costly misprediction outweighs the cost of executing both paths. We demonstrate that by guiding the existing if-conversion heuristics with this entropy information, we improve performance by up to 2.4% compared to the default implementation of this heuristic in the LLVM compiler. This work together with extensions on the ISPASS paper resulted in a publication in IEEE Transactions on Computers:

S. De Pestel, S. Eyerman, and L. Eeckhout. Linear Branch Entropy: Characterizing and Optimizing Branch Behavior in a Micro-Architecture Independent Way. IEEE Transactions on Computers (TC), 66(3) pages 458-472, 2017.

Contribution #4: RPPM: Micro-architecture Independent Multi-Threaded Performance Modeling.

To make the most out of the current multicore processor hardware software needs to adapt. To speed up applications, software developers can no longer rely on computer architects to speed up single-threaded performance. Instead, they need to adapt their applications and parallelize the work to utilize the available multicore hardware to the best possible extent.

Multi-threaded applications add yet another dimension for the software engineer and hardware architect to take into account. To enable the software engineers to adopt their applications to new upcoming hardware and to enable the hardware architect to evaluate new designs, new tools are needed. Existing simulation techniques lack the speed to simulate long-running applications across a large design space and do not always provide the necessary insight into the bottlenecks and issues of parallel applications.

We already developed a micro-architectural independent profiling tool for single-threaded applications, and this tool proved to be a useful complement to simulation when exploring a large design space. Extending this tool for multi-threaded applications requires changes to the memory model to accurately model interference in both private and shared cache levels. In addition, it requires a way to model synchronization overhead caused by synchronization primitives used by multi-threaded applications.

This new multi-threaded performance model was published in IEEE CAL and accepted for ISPASS 2019. In these publications, we demonstrate that with an average prediction error of 11% across the Parsec and Rodinia benchmarks, this is a valuable tool to speed up design space exploration and gain insight in the behavior of these applications on future designs.

> S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout. RPPM: Rapid Performance Prediction of Multithreaded Applications on Multicore Hardware. IEEE Computer Architecture Letters (CAL), 17(2), pages 183-186, 2018.

> S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout. RPPM: Rapid Performance Prediction of Multithreaded Workloads on Multicore Processors. IEEE International Symposium on Performance Analysis of Systems Software (ISPASS), accepted, March 2019.

1.4 Other Research Activities

During my PhD, I also took the opportunity of applying for an internship at Google Inc. I was fortunate to be one of the 2000 interns that got in and I started working at the GMail backend team, a group led by Amer Diwan, for 12 weeks. The main focus of my research was to build a framework that used empirical data to create a map of how requests are processed by the different backend structures of GMail and how they communicate with each other and the rest of Google's infrastructure.

When users access their email, requests are sent through the GMail system. Every now and then, the system selects a request and records how this request was handled by the backend and what subsystems this request needs to successfully handle the request. This data is stored and is later used by system architects to study the behavior of the system. But due to the massive amount of requests, GMail has to handle every minute of every hour of every day, and hence a highly optimized and parallelized framework had to be developed to translate this data into an insightful graph. This graph is automatically generated every day, and can be used to answer questions like: How many requests go to a specific subsystem? What is the average and tail latency of these requests? How many requests failed? Are we able to recover from this failure or does this lead to an error?

These massive data sets required massively parallel applications to process them. Building and optimizing these applications gave me valuable insight in how these applications work, and what the scaling bottlenecks are. When restarting my work on the micro-architectural independent tool to profile multi-threaded applications after the internship at Google, this new insight proved helpful into the development of the tool itself.

Although this internship did not result in any publications, these 12 weeks were a valuable experience both from a professional and personal standpoint.

1.5 Thesis Overview

Chapter 2 introduces the necessary background to understand the thesis. We discuss how superscalar out-of-order processors work, what type of instructions there are, how they flow through the different pipeline stages, and how they affect the overall execution time of a computer application. Next, we explain different simulation techniques and give an introduction to the interval model. Lastly, we explain the advantage of a micro-architactural independent modeling.

The remainder of this thesis provides an overview of the research that was done during my PhD. The thesis is organized in two parts. Part I discusses the research and modeling technique that we developed to model branch behavior of applications independently of the underlying micro-architecture.

Chapter 3 introduces our new Linear Branch Entropy metric which is a micro-architectural independent model to model the branch behavior of an application. We demonstrate that it is more accurate than previously proposed techniques such as taken rate, and transition rate, as well as the entropy model proposed by Yokota et al. [9, 24, 61].

Chapter 4 illustrates the usefulness of the Linear Branch Entropy model to analyze state-of-the-art branch predictors, to create a well-balanced set of benchmarks, to analyze the impact of history length, and to cluster benchmarks to speed up branch prediction simulation.

Chapter 5 shows how Linear Branch Entropy can be used to analyze the impact of mispredicted branches on the total execution time of an application. Chapter 6 is an extensive case study where the insight gained by a Linear Branch Entropy model is used to speed up an application. Through if-conversion, which is a technique that converts hard-to-predict branches into conditional move instructions to minimize the performance impact of mispredicted branches.

Part II of this dissertation describes the extension of the micro-architectural independent model to accurately model multi-threaded application performance on multicore processors.

Chapter 7 introduces this model with a thorough explanation of the two key challenges when modeling multi-threaded application, i.e., memory interference

and synchronization. We also show that we outperform naive extensions of the single-threaded model. Chapter 8 demonstrates that the model can be used to speed up design space exploration, and gain valuable insight in the execution behavior of a multi-threaded application by generating bottlegraphs.

Finally, in Chapter 9, we conclude the thesis and discuss avenues for future work to extend the model to new types of processor cores and emerging applications.

Chapter 2

Background

This chapter provides background information about superscalar out-of-order processors and how to create performance models for this type of processors. This chapter also provides some background on the interval model since this model forms the basis for the work done in this thesis.

2.1 Processor Architecture

The models we develop in this thesis are aimed at modeling superscalar out-of-order (OoO) processors. Therefore this section provides a brief overview on the way these OoO processors work and how they are able to speed up the execution of an application. We start by introducing in-order processors after which we build up to superscalar out-of-order processors [26, 36].

2.1.1 In-order Processor

A processor handles every instruction of the application it is executing in a number of steps before its completion. Figure 2.1 shows the pipeline through which every instruction needs to pass from left to right. These steps are:

Fetch: The fetch stage is the first stage in the pipeline and reads the instructions from memory. This unit also has to decide which instruction to fetch next. In general it will fetch the next instruction in the application binary, but instructions such as branches, jumps, calls, etc. are able to change the control flow of the application.

Decode: There are a lot of different instructions, therefore the next stage of the pipeline decodes the instructions. This stage reads the binary instruction, collects the data from the register file and communicates this to the execute stage of the processor.

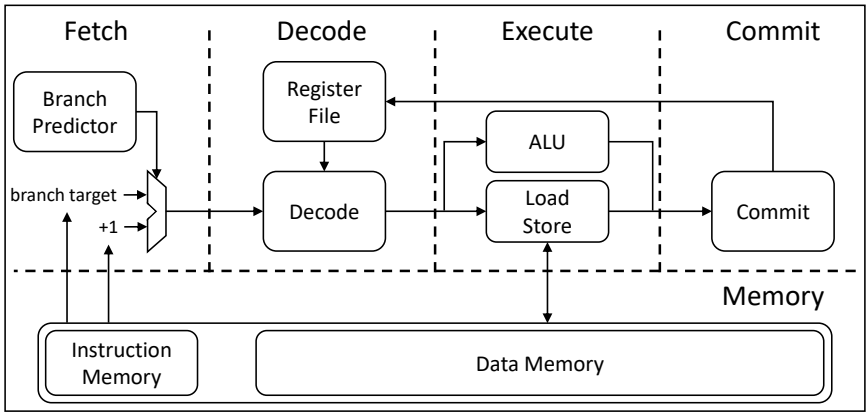


Figure 2.1: Pipeline model of an in-order processor.

The decode stage is the last stage of what is called the front-end of the processor pipeline. Every stage after the decode stage is part of the back-end of the processor pipeline.

Execute: Once the processor decodes the instructions, it can send the instruction to the correct execution unit and perform the actual operation. In the execute stage, a distinction is made between arithmetic instructions and instructions that communicate with memory. This is shown in Figure 2.1 with two units: one for arithmetic instructions (the ALU) and one for load and store operations from and to memory.

Commit: The last stage of the pipeline is the commit stage, in which the architectural state of the processor is updated with the result of the operation.

As mentioned before, there is a whole range of instructions that can be used. These instructions can be grouped in three different categories:

Arithmetic and logic operations

This first group contains the arithmetic and logic operations. These are the instructions that perform the actual work. This includes arithmetic instructions to add, subtract, multiply or divide numbers, compare operations to check if two values are the same, greater, smaller, etc., and bitwise operations such as xor, shift, etc.

These instructions are executed by the arithmetic logic units or ALUs in the execute stage of the pipeline (see Figure 2.1). The input for these operations is provided by the register file during the decode stage or by a constant value in the application executable file itself. At the end of the pipeline, during commit, the result is written back in a register in the register file, ready to be used by the next or subsequent instructions. Most of these instructions only take one

cycle[1] to execute, but more complex floating-point operations can take up to tens of cycles. These long-running instructions can cause a bottleneck in the execution of the application where the processor needs to stall to wait for the result.

Control flow operations

Control flow operations are instructions that change the control flow of the application. In the fetch, stage the processor fetches instructions in sequential order and sends them to the next stage, but the next instruction is not guaranteed to be the correct instruction to fetch. A branch or a call can interrupt the normal flow of instructions by redirecting the processor to another instruction instead of the next one.

A call or a jump is not difficult for the fetch stage to handle correctly since the address of the next instruction is either given as an argument, or predicted by the branch target buffer. But a branch, and more specifically a conditional branch, is more difficult for a processor to handle, since the address of the next instruction is only known after this branch is executed. This can cause a delay where the processor needs to wait for the branch to reach the execute stage before it can know what the next instruction is going to be. To avoid wasting time waiting for the branch to get executed, a branch predictor will provide a prediction for the next instruction (see Figure 2.1). If the predictor is able to correctly predict the outcome, the processor will be able to continue working without wasting time. If the prediction was wrong, the processor will need to stop, remove all incorrect instructions from the pipeline and start fetching again from the correct path.

Data handling and memory operations

The last group of instructions contains instructions that have to do with reading from or writing to memory. Processors have become faster with every generation, but the memory was unable to scale at the same rate, therefore in current systems, reading from or writing to memory takes a very long time for the processor. In the meantime the processor is forced to wait until the data becomes available before it can process this data. To solve part of this problem, a cache is introduced.

2.1.2 Caches

Accessing to main memory takes more than 100 processor cycles. This means that every time the processor wants to load data from memory it must wait for at least 100 cycles. Remember that most arithmetic instructions only take one cycle to execute, so 100 cycles is a very long time for the processor to wait. To reduce this long access time, a cache is typically placed in-between

[1] A cycle is the period of the processor clock.

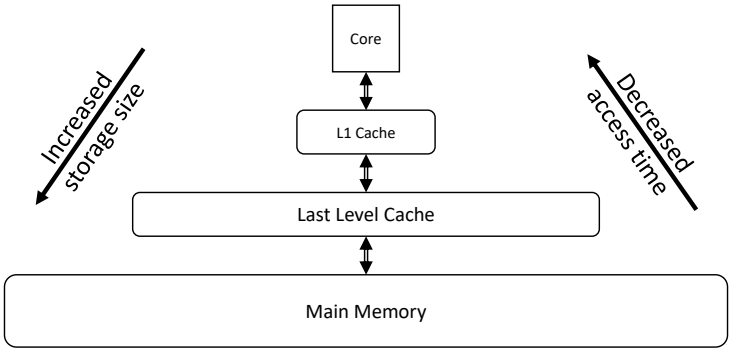


Figure 2.2: Model of a cache hierarchy where two cache levels were added in-between the core and main memory.

the processor and main memory. This is shown in Figure 2.2 where two cache levels are in-between the core and main memory. This cache has a fast access time, but can only hold a limited amount of data. The bigger the cache, the longer the access is going to take.

Therefore, a hierarchical structure is used with a fast but small cache at the first level and a slow but big cache at the last level. If the processor now wants to access data it will first go to the first level. If the data resides in this level, it will only take a couple cycles before the data is made available in the register file; if not, it goes to the next level, etc. In theory, the hierarchy can have as many levels as possible, but in most systems the hierarchy is limited to three levels of cache.

A cache can only hold a limited amount of data, therefore when the cache is full and new data is loaded, it needs to remove data in order to free space for the new data to be stored instead. A replacement policy is needed to select which data to remove. Using a least recently used or LRU replacement policy ensures that data that is frequently used is highly likely to stay in the cache. But when data needs to be removed, or when it is the first access to that data element (e.g., a cold miss), the likelihood of encountering an access to data that is not in the cache is still there. Since these accesses are unavoidable, a different solution needs to be made to further decrease the performance penalty due to these miss events.

2.1.3 Out-of-Order Processing

When waiting for a long-latency memory access, the processor cannot execute instructions that depend on the data that is needed from memory, but not all following instructions are necessarily waiting for this data. If there are instructions that do not need this data, it is possible to continue doing useful work by executing these independent instructions. Therefore we need to enable the processor to execute instructions out-of-order with respect to sequential program order.

This is done by introducing buffers in front of all ALUs and a data structure called a reorder buffer or ROB, which keeps track of all instructions that are currently present in the processor. The ROB is filled by the front-end pipeline and is used to keep track of how they originally entered the back-end stage of the processor. When the instruction at the end of the list (the oldest instruction) is finished, this instruction will be committed and removed from the ROB. Instructions enter and leave the ROB in program order, but execute on a functional unit possibly out of program order – hence the term out-of-order execution. Register renaming eliminates false data dependences (anti and output dependences) so that only real (read-after-write) data dependences remain, which enables data flow execution.

Out-of-order execution enables the processor to continue executing instructions as long as there are instructions left in the ROB that do not depend on outstanding memory accesses. When the processor keeps executing instructions, it is possible to execute a second long-latency load and end up with multiple outstanding memory accesses, leading to memory-level parallelism or MLP. By exploiting MLP, the processor can hide (part of) the long access latency of going to main memory by overlapping them, resulting in a significant decrease of the execution time.

2.1.4 Superscalar Out-of-Order Processing

Now that we can execute instructions out-of-order, there is a high probability that there are more instructions available that are ready for execution. The IPC or instructions per cycle indicates how many instructions the processor can execute in parallel on average. All the designs that were discussed so far could only execute one instruction every cycle, thus only reaching a maximum IPC of one. In order to increase the number of instructions that can be executed in parallel, **more ALUs** are needed leading to superscalar execution.

By increasing the number of resources, more instructions can get executed in parallel. But if we are able to execute more instructions in parallel, this means that we can finish all instructions in the ROB in a shorter time, therefore **the width of the front-end** needs to be **increased.**

By increasing the width of the fetch and decode stages, the processor can fill the ROB fast enough to keep the ALUs busy. The width of the front-end pipeline is the upper bound for the achievable IPC. It is not possible to achieve an IPC that is higher than the number of instructions the front-end pipeline can provide per cycle. However, to achieve a high IPC, enough independent instructions need to be available in the ROB ready to be executed, therefore **a bigger ROB and issue buffers** are needed.

A bigger ROB enables the processor to maximize the possible IPC and MLP by exposing more instructions to the ALUs. Together with a bigger ROB, bigger issue buffers are needed since all instructions need to fit in these buffers as well. As soon as the buffers reach their capacity, the front-end pipeline needs to halt issuing instructions even if the ROB is not full yet. But when increasing

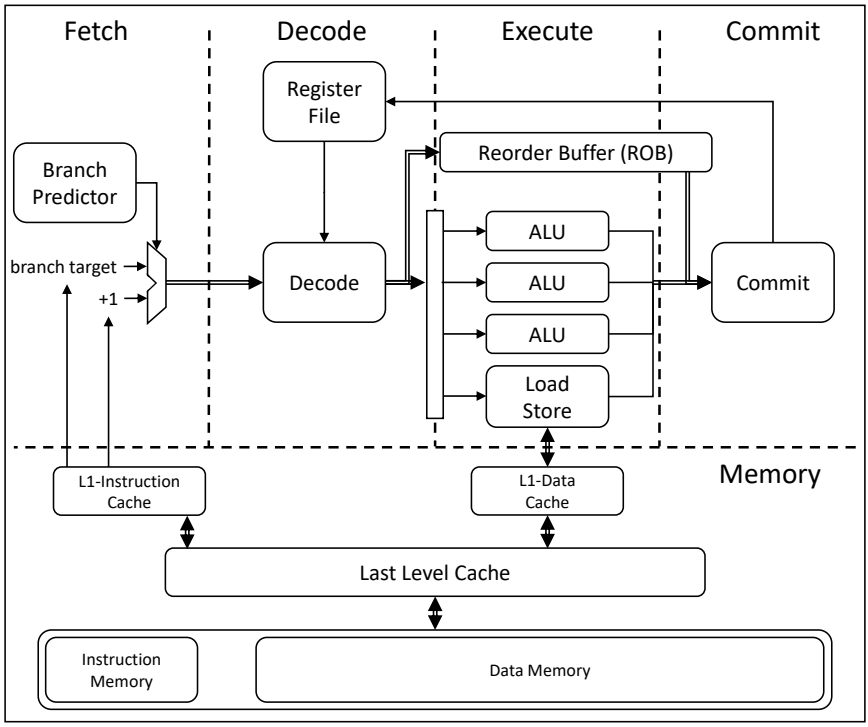


Figure 2.3: Pipeline model of a superscalar out-of-order processor with a wider front-end pipeline and bigger execute stage.

the ROB size and the rate at which we can fill the buffer, we need **to increase the commit width** as well.

To avoid the ROB from filling up with instructions that are finished we need to be able to commit finished instructions (at least) at the same rate as the front-end pipeline can fill the ROB with new instructions.

A superscalar processor is obviously more complex than suggested by Figure 2.3. A superscalar processor has multiple pipeline stages, and various components with specific configuration parameters. The entire design needs to be balanced to achieve the highest possible performance over a wide range of applications. To enable a computer architect to tune their design, there is a need for tools that provide a prediction of the performance of the aforementioned design without the need for an actual hardware implementation, because this would be very time consuming and extremely expensive.

2.2 Performance Modeling

Performance modeling can be used to evaluate designs in the early stages of the design cycle where building a prototype and then measuring the perfor-

mance of that hardware prototype is unfeasible. Performance modeling can be done using simulation [17] and/or analytical modeling [35].

2.2.1 Simulation

A simulator is the go-to tool for evaluating a new architectural design. The level of accuracy for cycle-accurate simulation is extremely high, hence it may appear to be the perfect tool for taking important design decisions. These simulators essentially implement the new design in a high-level programming language enabling it to simulate the new design on any machine and evaluating every aspect of the design in great detail.

There are a lot of different approaches to simulation: functional simulation versus timing simulation; trace-driven versus execution-driven simulation; a simulator can implement a complete system or only implement one aspect of the machine. All simulation techniques have their benefits and downsides. We now discuss the most important simulation techniques.

Timing Simulation

Timing simulation is used when the architect is most interested in timing-related design choices [17]. The result of a timing simulation will tell the user how long an application will run, if it will run correctly, when specific events happen and when possible performance bottleneck arise.

Timing or detailed simulation will simulate the application in a cycle-accurate way. Timing simulations simulate every aspect of the design, leading to very high accuracy but they are extremely slow since it will take a significant amount of time to simulate a single instruction, which would only take one cycle in real hardware, leading to a slowdown in the order of tens of thousands.

To avoid the high slowdown of cycle-accurate simulation, the level of detail can be tuned back. One alternative is cycle-level simulation [5, 7, 62]. Cycle-level simulation will only provide high detail in some parts of the design and do a fast and less accurate simulation of the rest of the architecture. This leads to a significant speedup compared to cycle-by-cycle simulation while still maintaining a high level of accuracy.

Functional Simulation

One way of speeding up simulation even more is through pure functional simulator [3, 42]. Functional simulation only simulates the functionality of the system, without a timing model. Therefore, this simulation does not provide all timing details of the system. This simulation technique only checks if the system works correctly and provides high-level statistics such as dynamic instruction count, cache miss rate, the number of mispredicted branches, etc.

High-level information such as the prediction error of the branch predictor can be measured by only simulating the branch predictor unit. Because it only simulates conditional branch instructions and only has to simulate the branch predictor unit, this will decrease the simulation time significantly compared to a detailed timing simulation of the complete system.

Trace-Driven Simulation

Trace-driven simulators are using a sequential instruction trace as the input to the simulator [17]. The information in the trace will depend on what the focus of the simulation is. If the simulator only simulates the cache, it only needs a trace containing all accesses to memory. But when it is doing a detailed complete system simulation, the trace will need to contain information for all dynamically executed instructions.

This can lead to problems when simulating long-running applications, since the dynamic instruction count executed by these applications can be very high, and the file containing this trace can become too big to store [22]. Therefore some simulators are adapted to generate the trace on the fly by using a tracer/profiler that is running and feeding the trace to the simulator as it is simulating the trace, thus eliminating the large storage need.

Even if the simulator is using a trace from a file, this file is first generated using a tracer or a profiler. Most of these only record instructions that were successfully completed and lack instructions executed on the wrong path after a branch misprediction. Therefore these traces are not always representative of the real execution of an application.

Execution-Driven Simulation

To solve the problem of the large traces and the lack of information about instructions on the wrong path, the actual static binary can be used as an input to the simulator. This is called execution-driven simulation [17]. Since this is not a trace file but a file only containing static instructions, every instruction appears only once in the file, thus significantly reducing the file size.

Since these files also contain instructions that are not necessarily getting executed, including instructions from both paths of a branch, it is very easy to simulate wrong-path instructions in case of a branch misprediction.

Sampled Simulation

Instead of simulating the complete execution of an application, several techniques were proposed to speed up simulation by only simulating a sample of the complete application execution and extrapolating the results to the complete execution [17].

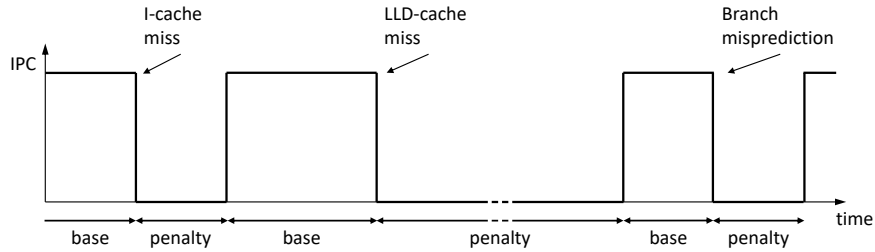


Figure 2.4: IPC shows an on/off behavior when focusing on the dispatch stage.

There is a whole range of possible sampling techniques. A pure *random sampling* technique selects multiple regions at random and simulates these in detail [15]. A second method of selecting the regions to simulate is one where the regions are *distributed evenly* throughout the whole program execution [59]. These methods do not check whether the selected regions are representative for the complete execution of the benchmark, instead they rely on the idea that as long as you select enough regions throughout the entire execution, it should be statistically representative.

In order to have a higher guaranteed accuracy a pre-run can be used, during which a *profile* of the application is built [53]. This profile is then used to select the most representative regions of the applications to simulate in detail.

2.2.2 Analytical Modeling

When systems become too large or applications execute too many instruction for detailed simulation, an analytical performance model can be used to approximate performance metrics [34, 35]. These analytical models rely on probabilistic methods, queuing theory, etc. They replace detailed simulation with simple formulas that provide insight into how an application behaves on the architectural design that is being evaluated.

Analytical models are extremely fast because they are based on simple mathematical formulas and algorithms, and building these models is also a lot easier than developing a complex detailed cycle-accurate simulator. The input to these models are statistics such as instruction dependencies, cache miss rates, branch misprediction rates, etc. that were obtained through specialized functional simulation.

To build these models, some assumptions are made to keep the model simple, but this does not necessarily lead to low accuracy as we will demonstrate later in this work. Instead, by keeping the models simple, the insight gained from these models is higher than what can be gained from detailed simulation.

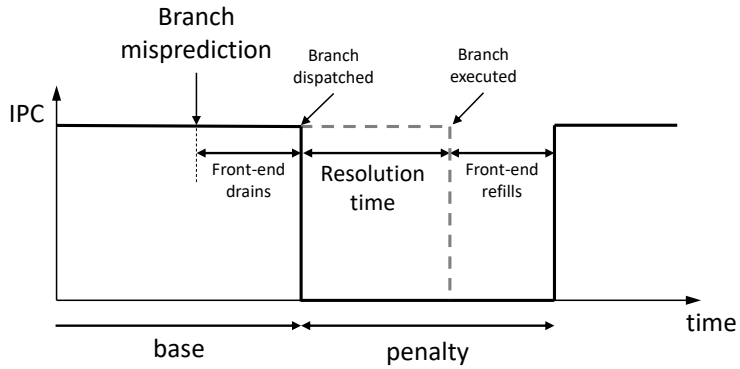


Figure 2.5: Detailed view of the interval timing of a branch misprediction.

2.3 Interval Model

Interval analysis [20] in an intuitive model that is based on the observation that, without miss events, the performance or instructions per cycle (IPC) of the processor pipeline should be equal to the width of the pipeline. This means that the processor is able to process instructions at its maximum IPC until a miss event occurs. This is illustrated in Figure 2.4, where IPC is maximized in-between miss events at the designed dispatch width. Every time a miss occurs, IPC drops to zero since no instructions can be processed by the pipeline any further.

This behavior is best observed when looking at the dispatch stage, which is the pipeline stage at which instructions leave the decode stage and enter the issue queue and ROB before being executed. At the fetch stage, the IPC can show peaks and dips when reading non-sequential instructions, for example at a branch or a function call, because these instructions reside most likely in another cache line. Therefore, IPC will drop to the maximum number of instructions that can be loaded from that cache line and ramp up the next cycle to make up for lost time. At the execute and commit stages, the periods of zero IPC can be hidden by out-of-order execution of instructions that are left in the ROB and are not on the critical path of the miss event. This makes any of these stages (fetch, execute, commit) less ideal for interval analysis.

The reason for and the length of the penalty is different for every miss type, as we will describe in the following paragraphs.

Branch misprediction

When a branch is fetched, it is not clear what the next instructions are going to be. A branch can interrupt the normal flow of instructions and can send the execution of the application to another part in the code. But the processor can only know this after the branch is executed. Therefore waiting is not an option, as it is better to predict the outcome of the branch and start fetching

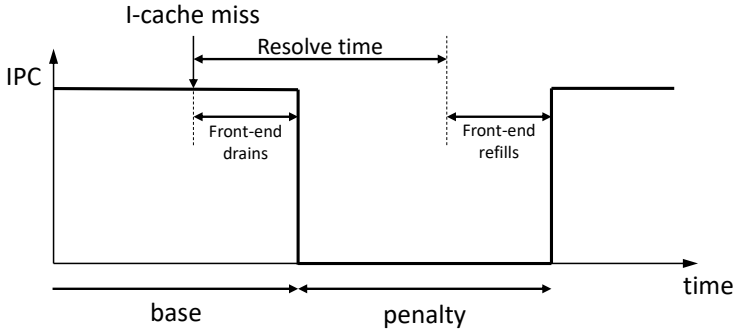


Figure 2.6: Detailed view of the interval timing of an I-cache miss.

these instructions. In case the branch predictor made the correct prediction, the processor does not stall, leading to a significant speedup.

But when the branch predictor made an incorrect prediction, all instructions fetched after this branch just became useless and need to be removed from the pipeline. This is visualized by the dashed line in Figure 2.5. During the first part of the penalty, the processor was still fetching and dispatching instructions so the IPC was still high. But the processor was actually fetching and dispatching instructions that would later be removed from the ROB, so the effective IPC was actually zero.

The time it took for the processor to realize that it was handling instructions on the wrong path is called the branch resolution time. This is the time it takes the branch to reach the execution stage after being dispatched. This time depends on the length of the dependency path of the branch, since all instructions on the dependency path need to be executed before the branch is executed and is thus not a constant value. After the branch resolution time, the front-end needs to fill up again before useful instructions are dispatched. The sum of the branch resolution time and the front-end refill time is the total penalty for a branch misprediction.

I-cache miss

An I-cache miss happens at the fetch stage of the pipeline when the processor is not able to fetch the next instruction from the instruction cache. If this happens, the processor will need to fetch the instructions from a lower cache level, or in the worst case, read the instructions from memory.

Now that the fetch stage is not able to provide the next instruction, the decode stage will soon stop dispatching instructions, and IPC will drop to zero. There is a delay between the miss event and the moment at which the dispatch rate drops to zero. The reason for this delay is because the miss happens right at the start of the pipeline, but dispatch only happens when instructions leave the front-end pipeline, so all instructions that are successfully fetched but not yet dispatched can still continue to the execution stage. After this delay, the

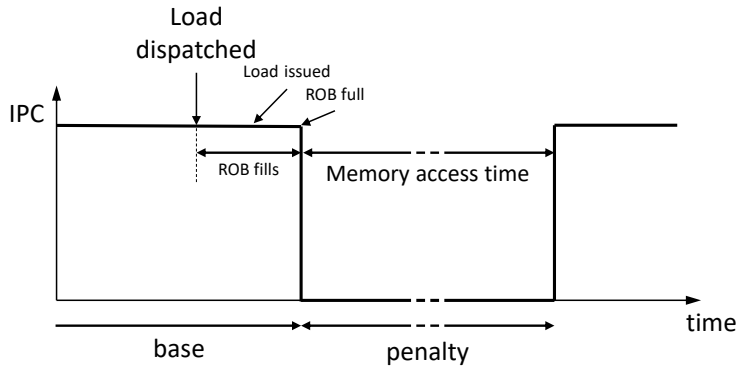


Figure 2.7: Detailed view of the interval timing of a long latency data cache miss.

dispatch rate will stay zero until the miss event is resolved and the front-end pipeline is filled up again.

Figure 2.6 illustrates this. Dispatch can continue dispatching instructions until the front-end is empty and will restart after the miss is resolved and the front-end is filled again. Since the time it takes for the front-end to drain and refill is the same, the penalty is not influenced by the length of the front-end pipeline. Therefore the penalty is equal to the time it takes for the miss event to be resolved.

Data cache misses

When a load is issued, the processor can continue dispatching and executing independent instructions. As a result, the small latency of going to the first levels of cache can be hidden. But the processor can only continue processing instructions until the reorder buffer fills up, at which moment dispatch halts and waits for empty slots in the reorder buffer. Therefore, the model can be simplified by only taking accesses to main memory into account, assuming that the access latency to the cache is small enough to be hidden by the ROB. This is in contrast with the I-cache miss where even a small latency cannot be hidden.

This is illustrated in Figure 2.7. IPC at dispatch will only drop to zero when the ROB is full. And IPC will ramp up again when the data returns from main memory and the load can be committed and removed from the ROB.

When a store is issued and the (old) data is not in the cache, this also produces a long-latency access to memory, but since instructions directly following a store will (almost) never depend on that store, we can buffer the stores and commit the stores even before the data is actually written to memory. When a load is issued, we can first check this store queue.

Overlapping misses

Figures 2.6, 2.5 and 2.7 only show what happens with an isolated miss event. Front-end miss events will never overlap each other. When an I-cache miss occurs there are no other instructions in the pipeline to fetch, hence there are no branches to predict. If the processor is fetching instructions along the wrong path, after a branch misprediction, an I-cache miss will not influence performance since instructions fetched were not useful anyway.

But since the processor is able to continue fetching and executing instructions that are not relying on the data it is waiting for, it is possible for the processor to encounter a second data cache miss in the same ROB. When this happens, the latency of the second load is completely hidden underneath the first load. The number of overlapping misses if at least one is outstanding is called MLP or memory-level parallelism. This was already discussed as one of the main performance benefits from out-of-order processing of instructions. Therefore, in order to have an accurate estimation of the performance impact of long-latency memory accesses due to cache misses, we will need to take MLP into account.

Overlapping events between front-end and back-end misses do occur, and are more likely with increasing ROB size, but the impact on overall execution time was found to be small enough to be left out of the model to keep the model simple.

Resulting model

$$C = \overbrace{\frac{N}{D}}^{\textbf{Base}} + \underbrace{m_{bpred} \times (c_{res} + c_{fe})}_{\textbf{Branch}} + \overbrace{\sum_{level=i} m_{ILi} \times c_{Li+1}}^{\textbf{I-cache}} + \underbrace{\frac{m_{LLC} \times c_{mem}}{MLP}}_{\textbf{D-cache}} \quad (2.1)$$

The resulting model is described by Equation 2.1. The first component is the **base** component and estimates the execution time without any miss events by dividing the number of dynamically executed instructions (N) by the width of the front-end pipeline (D). The second component calculates the impact of all incorrect **branch** predictions (m_{bpred}) by multiplying them with the sum of the average branch resolution time (c_{res}) and the length of the front-end pipeline (c_{fe}). The third component is the impact of **instruction cache** misses. As mentioned before, even the small latency of going to a level-2 cache influences the execution time, so the accesses to all cache levels will need to be counted (m_{ILi}) and multiplied by their respective access penalty (c_{Li+1}). For the last component, only the accesses to main memory due to **last-level cache load** misses (m_{LLC}) are counted and multiplied by the access time to main memory (c_{mem}). But since here overlapping accesses will decrease the

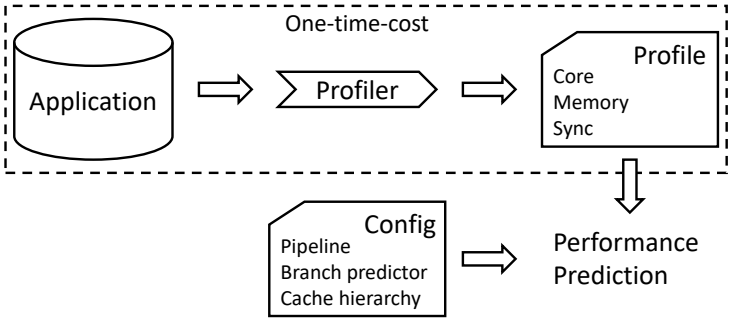


Figure 2.8: Schematic overview of how micro-architectural independent modeling works.

impact on overall execution time, this component is divided by the average memory-level parallelism (MLP).

2.4 Micro-architectural Independent Modeling

The interval model provides a performance prediction much faster than simulation, because we avoid the need for a time-consuming and complex timing simulation. Instead, we are able to compute all inputs to Equation 2.1 by some simple and lightweight functional simulations. A branch simulator calculates the number of incorrect branch predictions for a specific branch predictor. A cache simulator provides cache miss numbers for all levels of the I-cache and the total number of main memory requests again for a specific cache simulator. Finally, an ROB simulator estimates the average branch resolution time and MLP.

Running three functional simulators and combining all results in a simple mathematical equation is faster than running a timing simulation, but this process is a recurring cost for every new design a computer architect wants to analyze. Although when trying a new branch predictor, the cache numbers are not likely going to change, and the ROB simulator could calculate the numbers for a different set of ROB lengths at the same time, still the overhead for analyzing and predicting performance for a large design space is significant.

Instead of re-running the functional simulations for every design point, a micro-architecture independent profile could be build [58]. Using this profile, we can estimate the inputs to Equation 2.1 by combining this micro-architecture independent profile with the design or designs we want to analyze. Figure 2.8 gives a schematic overview of how this would work. On top, the building of the microarchitecture-independent profile is shown: this is done by a profiling tool and is a one-time cost per application. The second step is to combine this profile with the processor configuration of interest to predict performance using a mathematical model like the interval model. Using this approach reduces the time needed for a design space exploration significantly because the profile

needs to be computed only once after which it can be re-used for predicting performance across a range processor configurations in the design space.

Some of the inputs needed for the micro-architectural independent model are easy to profile. The number of successfully committed instructions can even be counted using hardware performance counters; the instruction mix can be profiled using an instrumentation tool and could give an estimate of the average instruction latency. StatStack is a tool developed specifically to profile the memory behavior of applications in a micro-architecture independent way [18]. Some prior work has been done to model the branch behavior, although a new tool is needed to enhance its accuracy. A model for MLP, issue stage contention, etc. are also needed. A more in depth explanation about how to model single-threaded and multi-threaded application performance is provided in Part II.

Part I

Linear Branch Entropy

The first part of the thesis describes how linear branch entropy is defined and how it can be used to model branch behavior of applications, how it is used to estimate the overhead introduced by the branch predictor when modeling the execution time of an application and how it helps steering if-conversion to optimize performance.

Chapter 3

Linear Branch Entropy

3.1 Introduction

Branch prediction is and will remain an important performance contributor. Branch mispredictions disrupt the continuous flow of instructions in a core, and although advanced branch predictors succeed in correctly predicting the majority of branches, the impact of branch mispredictions on overall performance is non-negligible for many applications. Therefore, performance analysis and estimation techniques should take into account branch prediction effects.

Previous work has quantified the penalty of a branch misprediction [20, 21], i.e., how many cycles are lost when an incorrect prediction occurs. However, to the best of our knowledge, there currently exists no technique to profile branch behavior in a branch predictor independent way. The common technique to analyze the performance impact of branch behavior is to simulate a particular branch predictor for a particular application. This is not scalable if the number of applications and/or the number of possible branch predictor configurations is large, as is the case for example in a large design space exploration study. All different branch predictors need to be simulated for all applications, resulting in a large number of simulations. Furthermore, the simulator only provides the misprediction rate and gives no insight into the underlying causes: is a high miss rate caused by the irregular behavior of the branches in the application, or by the predictor performing poorly, or both? A micro-architecture independent characterization analyzes each application only once, obtaining an application-specific – but branch predictor independent – profile. This profile is then used to estimate the number of branch mispredictions for this application for different types of branch predictors, without simulation or re-profiling. To be meaningful, a predictor-independent branch profile should have a good correlation with the branch misprediction rates of specific branch predictors. This is not straightforward, because branch prediction accuracy depends both on particular branch patterns in the application and the specific structure of the predictor. Given the multitude of different branch predictors, each having

different patterns that they can or cannot detect, finding a single profile that correlates well for all is challenging.

We propose *linear branch entropy*, a novel metric that quantifies how regular the branch behavior of an application is. An entropy of 0 means that the branches are highly predictable, resulting in few branch mispredictions; on the other hand, an application with an entropy close to 1 has a large number of branch mispredictions. We define *global entropy* as the entropy based on global branch history, and *local entropy* as the entropy based on local branch history. Furthermore, there is one entropy number per history length. All of these entropies can be measured through a single profiling run of the application of interest.

Linear branch entropy can be used to characterize and classify applications depending on their branch behavior. Furthermore, because entropy correlates well with misprediction rate, we can construct a model for a specific branch predictor that translates entropy into misprediction rate. We find that a linear relationship between entropy and misprediction rate fits best and is intuitive to understand. We therefore use a training set of benchmarks, for which we know both the entropy and the misprediction rate, to fit a line and use that model to estimate the branch misprediction rates for other applications. In summary, we have one entropy profile per application and one miss rate versus entropy model per branch predictor (two parameters to define a linear relationship). Combined, the misprediction rate for all applications and all predictors can be estimated. The linear model itself can also be used to analyze and compare branch predictors (without the input of specific applications): the two parameters of the linear model (constant factor and slope) indicate how well a specific branch predictor predicts regular (low-entropy) and irregular (high-entropy) branches.

We make the following novel contributions:

- We propose linear branch entropy, which correlates better to branch misprediction rates than the classic definition of Shannon's binary entropy, because it more closely resembles the organization of a branch predictor.
- We show that linear branch entropy indeed has a better correlation: we find that there is a linear function between entropy and misprediction rate, and we show that this function results in more accurate branch misprediction rate estimations than prior work (on average 38% lower error compared to the best prior work, i.e., a combination of taken and transition rate).
- We use linear branch entropy to classify benchmarks based on their branch behavior, and we show that we are able to obtain the same branch misprediction rate ordering by evaluating only 5 representative benchmarks out of a set of 40 benchmarks.
- We compare the top-four branch predictors of the latest branch competition using the new model, and show that the third runner-up is better

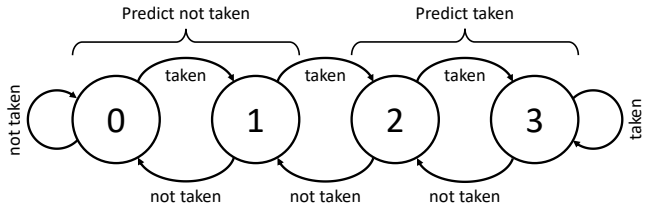


Figure 3.1: A state diagram of a two-bit saturating counter.

at predicting high-entropy branches than the winner. However, most of the evaluated benchmarks contain only low-entropy branches, and the winning predictor performs better for those branches.

Predictor-independent branch profiling is a key component of a micro-architecture independent performance model, enabling performance estimation and analysis over a large design space with a single profile. We envision the proposed microarchitecture-independent branch prediction model as a key part towards a complete performance model using microarchitecture-independent metrics only.

We begin by discussing prior work on branch classification and predictor analysis in Section 3.2. In Section 3.3, we then introduce our new entropy metric. We explain how to build the profiler and the branch predictor model, in Section 3.4 and evaluate the correlation of linear branch entropy to branch misprediction rates for different predictors and we compare against prior work in Section 3.5.

3.2 Background

In this section, we discuss the previously proposed taken rate and transition rate to classify branches, and discuss related work on using entropy to characterize branches. We also show the similarities between our way of measuring entropy and prior work in analyzing branch predictors as a Markov predictor. We begin by summarizing the basics about branch prediction.

3.2.1 Branch Prediction

Branch predictors predict the outcome of a conditional branch (and also the branch target address, which is not in the scope of this work). They do that by using information from the past, e.g., whether the branch has been previously taken or not. This information is represented by a saturating counter. Figure 3.1 depicts a two-bit saturating counter as a state diagram. Every time a branch is taken, the counter increments or saturates on three. When the value is two or three, the predictor will predict taken. When the previous branches were not taken, the saturating counter will decrease. Once the value drops to one or zero the predictor will predict the next branch as not taken.

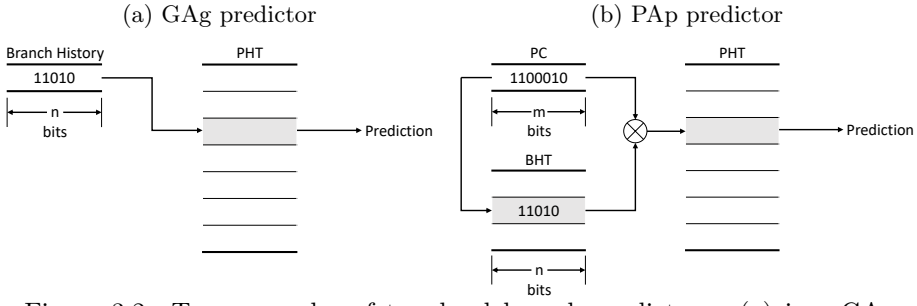


Figure 3.2: Two examples of two level branch predictors. (a) is a GAg predictor using global history to index the PHT. (b) is a PAp predictor, using both the branch address to index the history table and concatenates the local history with the branch address to index the PHT.

A bimodal predictor [55] uses a table of saturating counters, indexed by the program counter (PC) or branch address. This way it can predict the outcome of a branch based on the previous outcomes of that same branch. Because there are correlations between the outcome of the current branch and the outcomes of previous branches, branch predictors often keep track of history information. These predictors are called two-level predictors [60], because they use history at the first level, to index a pattern history table (PHT), at the second level.

In Figure 3.2 two examples of two-level branch predictors are shown. The first predictor is the GAg predictor (Figure 3.2a). The first level of this predictor uses a global history register, containing the outcome of all prior branches, to index the PHT. The second example is the PAp predictor (Figure 3.2b). This predictor records the previous outcome of each static branch instruction separately (per-address history), for which it uses the PC or branch address to index its branch history table (BHT). Next, it concatenates the branch history with the branch address to index the PHT.

These techniques can lead to more combinations like GAp (global history, per-address indexing) or PAg (local history, global indexing). A gshare predictor is a variant of a GAp predictor, which XORs global history and branch address bits, instead of concatenating them. A tournament or hybrid predictor combines multiple predictors, and uses a metapredictor to choose which predictor performs best for which branch instruction [45].

Modern state-of-the-art predictors, such as TAGE [51] and neural predictors [31], increase the history length without increasing the size of the predictor. The TAGE predictor does this by only increasing the length of the history for branches that benefit from this longer history. Easy-to-predict branches do not need this longer history, so by saving on hardware cost and warmup time for these branches, TAGE can select some branches and record a longer history for them. Classic two-level branch predictors grow exponentially in size with increasing history size. Neural branch predictors on the other hand scale linearly with growing history length, enabling them to increase the history significantly without the large hardware cost.

3.2.2 Taken Rate

Taken rate is defined as the number of times a given branch is taken divided by the total number of times the branch is executed [9]. Branches with a high (close to 1) or low (close to 0) taken rate are easy to predict, because they are highly biased towards a particular direction (taken or not taken). A taken rate of around 0.5 typically indicates a branch that is difficult to predict, because its outcome varies between taken and not taken. Taken rate can detect branches that are easy to predict, but classifies some branches that are easy to predict with some history information (e.g., a periodic switch between taken and not taken) as difficult to predict (because its taken rate is close to 0.5).

3.2.3 Transition Rate

To solve this problem, Haungs et al. [24] propose to measure transition rate to classify branches. Transition rate is defined as the number of times a given branch switches between taken and not-taken over the total number of times the branch is executed. A low or high transition rate indicates a highly predictable branch: a low transition rate means that the branch has a bias towards a certain direction, and therefore comprises both high and low taken rate branches. A high transition rate indicates a branch that switches frequently, and might therefore have a regular pattern that can be recognized by a predictor that keeps track of branch history.

The authors also show that a combination of taken and transition rate has a slightly better correlation with misprediction rate than each of the metrics separately. An important advantage of taken and transition rate is that they are very easy to measure and are independent of the particular branch predictor, i.e., they capture a program characteristic. However, they cannot detect branches that have a regular but slightly more complex pattern (e.g., a repeating pattern of two times taken and one time not taken has a taken and transition rate of 67%, but can be accurately predicted with two bits of history).

3.2.4 Branch Entropy

Yokota et al. [61] measure the entropy of branch outcomes using the standard Shannon entropy formula from information theory:

$$E = -\sum_i p(S_i) \log_2 p(S_i) \tag{3.1}$$

S_i denotes all possible branch outcome patterns and $p(S_i)$ the probability for pattern S_i. Yokota et al. define local and global history entropy, as well as predictor-dependent entropy. They show that entropy correlates well with misprediction rate and that inverting the entropy function to a binary probability (i.e., solving Equation 3.3 to p) yields an upper bound for branch prediction

accuracy. We find that a linearized version of entropy leads to a simpler model to estimate branch misprediction rates, and we are able to predict the actual misprediction rate of a specific predictor, instead of a lower bound.

3.2.5 Branch Predictor as a Markov Predictor

Chen et al. [12] show that a branch predictor, in fact, implements a simplified prediction-by-partial-matching algorithm, which is a set of Markov predictors of different orders. A Markov predictor of order m predicts the outcome of the next branch as the most frequent outcome seen in the past after the same outcome history of the last m branches. Our way of profiling is similar to building a Markov predictor: we collect the distribution of the outcome of a branch per history of m previous branches. This distribution is the input for our entropy metric. We then reduce the history by one bit to calculate the entropy for $m - 1$ history bits, similar to a partial matcher, which also reduces the order to find new patterns. However, we use this information to characterize branches and estimate branch misprediction rates, while Chen et al. use this insight to propose better branch predictors.

3.2.6 Micro-Architecture Independent Characterization

Several researchers use micro-architecture independent metrics to characterize and analyze programs. Joshi et al. [33] use taken rate and forward branch taken rate to characterize branch behavior, while Hoste and Eeckhout [27] use the misprediction rate of a theoretic prediction-by-partial-matching algorithm. Shao and Brooks [52] show that application profiling depends on the ISA, and present an ISA-independent profiler. They use the entropy model of Yokota et al. [61] to characterize branch behavior.

3.3 Linear Branch Entropy

The general idea behind branch prediction is that there exists correlation between branch outcomes. Depending on the outcome of previous branches or previous outcomes of the same branch, a particular branch can have a higher probability to be taken or not[1]. The better the correlation is between the outcome of the current branch and the history of previous outcomes, the more accurately the branch is predicted. The accuracy of a branch predictor thus depends on how stable the outcome is for each history pattern. We use this insight to quantify the predictability of a branch: for each branch i and history pattern H (which could be local or global history), we record the number of taken and not-taken branches, denoted $n_1(i, H)$ and $n_0(i, H)$, respectively. We

[1] We only consider branch predictors that use the history of outcomes of previous branches to predict future branches. The methodology can also be applied on branch predictors that have other inputs (e.g., call stack depth), by defining corresponding patterns.

then define the probability for a taken branch, given a specific history pattern, as

$$p(i,H) = \frac{n_1(i,H)}{n_0(i,H) + n_1(i,H)}. \tag{3.2}$$

This is similar to the definition of taken rate [9]. The main difference is that we do not calculate a single taken rate per static branch instruction, but a taken probability for each possible history pattern of each static branch.

Now we have to transform these taken probabilities into a branch predictability metric. As discussed before, a taken probability of 0 (never taken) and 1 (always taken) are highly predictable. An often used metric in physics and information theory to denote the amount of disorder or the amount of information in a system is entropy. The classic definition of binary entropy (also denoted Shannon entropy) is

$$E(p) = -p \times \log_2(p) - (1-p) \times \log_2(1-p). \tag{3.3}$$

Entropy is 0 if $p = 0$ or $p = 1$, and 1 if $p = 0.5$. This definition is used by Yokota et al. [61]. They showed that the inverse of this function, i.e., reconstructing p given $E(p)$, forms an upper bound for branch prediction accuracy.

We use a alternative definition of entropy, which we call *linear branch entropy*, defined as

$$E_L(p) = 2 \times \min(p, 1-p). \tag{3.4}$$

This equation also equals 0 for $p = 0$ and $p = 1$, and 1 for $p = 0.5$ (hence the factor of 2). It is easy to calculate, and we find that misprediction rate is approximately a linear function of this entropy, resulting in a simple linear model for the branch predictor's misprediction rate. The intuition behind this is that a branch predictor also performs a simple function: it selects the outcome with the highest occurrence in the recent past, by making use of (2-bit) saturating counters. Therefore, it is easy to see that misprediction rate is proportional to the probability of the least frequent outcome, hence the minimum of p and $1-p$. On the other hand, Shannon entropy relates to information theory, and reflects the number of bits needed to represent a certain amount of information, which is conceptually more complex than the functioning of a branch predictor. A limitation of the linear entropy function is that it is not differentiable in $p = 0.5$, which could be a problem in mathematical optimization, but which is not a problem for our model.

Figure 3.3 plots the Shannon binary entropy, based on Equation 3.3, and the new linear branch entropy, based on Equation 3.4, in the same graph. The Y-axis shows the entropy as a function of the probability of a taken branch on the X-axis. Both curves share the following characteristics: entropy equals zero if the probability is very low (0) or very high (1), and 1 when the probability equals 0.5.

After calculating the entropy for each branch and for each history pattern using Equations 3.2 and 3.4, we average the entropies over all branches and all history patterns. Let $n(i,H)$ be the number of occurrences of branch i with

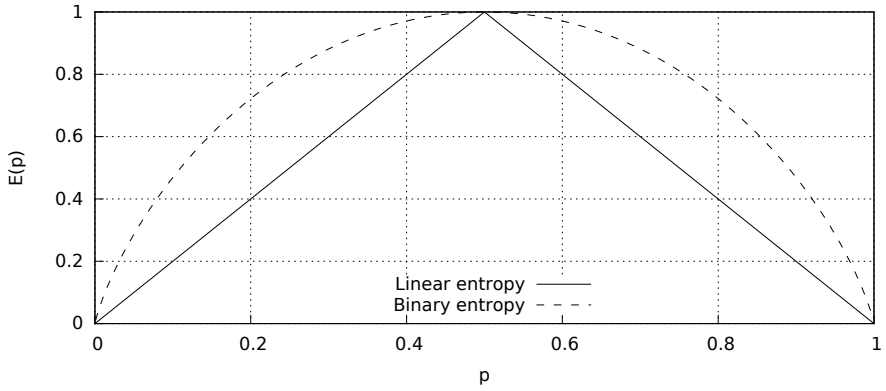


Figure 3.3: Comparison between Shannon's binary entropy and the linear branch entropy we propose.

history pattern H (i.e., $n(i,H) = n_0(i,H) + n_1(i,H)$), and let N be the total number of dynamically executed branches (i.e., $N = \sum_i \sum_H n(i,H)$). The average branch entropy of an application is then defined as

$$E = \frac{1}{N} \sum_i \sum_H n(i,H) \times E_L(p(i,H)). \tag{3.5}$$

This averaging method is sound, because branch entropy is linear in p. This is another advantage over the Shannon entropy metric, which cannot be easily averaged due to the logarithms, and which also explains the detour Yokota et al. [61] had to make to find an upper bound for the hit rate (calculating entropy over all patterns, and then solving this entropy for a binary probability).

3.4 Branch Predictor Model

In this section we discuss our three main components of the proposed branch predictor model: (1) we propose a branch behavior profile that is application-dependent but independent of a specific branch predictor, (2) we build a model that relates branch entropy to the misprediction rate for a specific branch predictor organization, and (3) we propose a fast branch predictor design space exploration tool by combining the application-dependent profile and the predictor-dependent model. This section elaborates on each of these three components. For the ease of discussion, we begin by explaining how the design exploration tool (third component) is organized, because this provides a clear overview on how the components relate to each other.

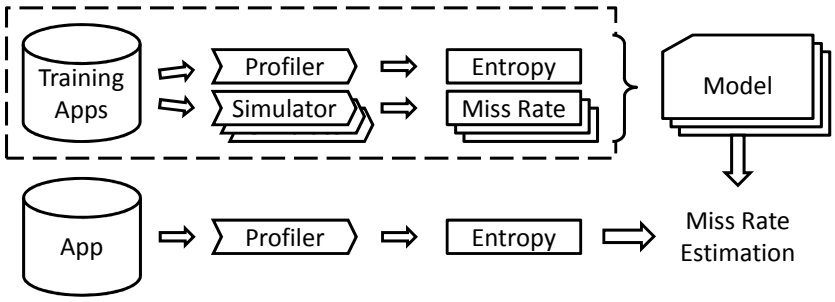


Figure 3.4: Schematic overview of how the branch predictor model works. The dashed box represents the training phase, which has to be done once for each branch predictor type.

3.4.1 Overview

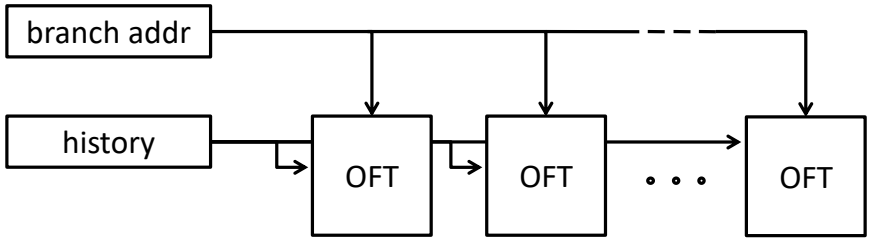
Figure 3.4 gives a high-level overview of the proposed framework. The calculation of the branch entropy is illustrated at the bottom: the application is executed within a profiler, resulting in a (set of) entropy number(s). The profiler is discussed in Sections 3.4.2 and 3.4.3. The top part of the figure shows the predictor-dependent model. The dashed box shows the steps that need to be done only once for each predictor to construct the model that relates branch entropy to misprediction rate. We first need to select some benchmarks for training. For this training set, we measure both the entropy and branch predictor accuracy. The branch misprediction rate is measured using a simulator. A simulation for every different type of branch predictor that we want to model is required, hence the multiple simulations in Figure 3.4.

The entropy and branch misprediction rates for the training set are then used to construct a model for each of the branch predictors. There is one model per branch predictor, which takes as input the branch entropy of an application and predicts branch misprediction rates. A branch predictor model is tied to a specific branch predictor, and can be used to estimate the misprediction rates for all applications.

Once the models are built, the misprediction rates for a new application can be estimated by profiling that application once, obtaining its branch entropy number. This entropy number is then used to estimate the misprediction rates for all modeled branch predictors instantaneously, by just evaluating a linear function. Introducing a new type of branch predictor requires simulating this branch predictor for the training benchmarks, and constructing a new model. Afterwards, this model can be used for all applications, for which we can reuse their already measured entropy numbers.

In summary, (1) we have to profile an application to measure its branch entropy to serve as input for the model to make a prediction, and (2) we have to construct a model using entropies and misprediction rates from a training set. The profiler itself also consists of two phases: recording branch histories

(a) Global History



(b) Local History

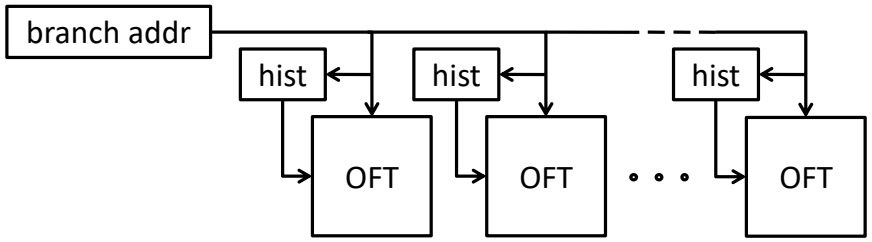


Figure 3.5: The profiler records branch outcomes and history of every unique branch, both for the global and local history.

and outcomes, and calculating entropy. The next sections discuss each of these steps.

3.4.2 Recording Branch Behavior

We need a profiling run to determine the outcome (taken or not) of every branch and record the branch history. We do this by maintaining a outcome frequency table (OFT) that is indexed by a history pattern, and that has two counters per entry: the number of not-taken branches n_0 and taken branches n_1 when this history pattern is encountered.

We keep two versions of the tables, one indexed by global history (outcome of all previous branches), and one indexed by the local history (outcome of the previous occurrences of the same static branch instruction), see Figures 3.5a and 3.5b, respectively. We make this distinction because current branch predictors use either local or global history, or a combination of both, and it is impossible to reconstruct global history from local history information or vice-versa without detailed information on how branches interleave.

Furthermore, we keep track of one OFT per static branch instruction, to be able to model branch predictors that use branch address bits to index the second level of the branch predictor. The history pointers keep track of a history of m bits, which is the largest history we want to model. From this, we can extract the entropy for all smaller history lengths, as we will explain in the next section.

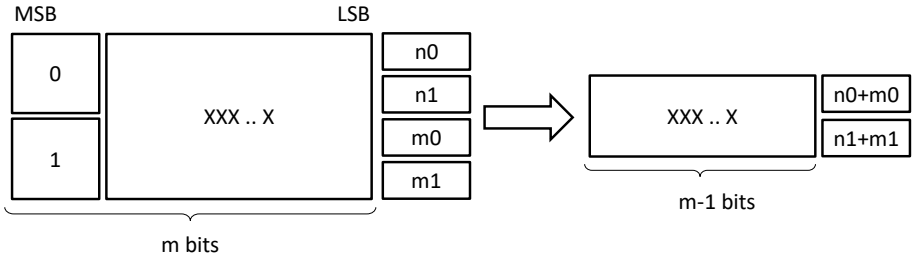


Figure 3.6: Reducing the size of the OFT to calculating branch entropy for a smaller history size.

Although this seems a lot of data, the tables are actually very sparsely occupied. Each branch has only a limited number of actually encountered history patterns, so only a fraction of the 2^m entries is used. Therefore, we use an associative array instead of a full table to record the information. In all of our profiling runs, our tool used at most 10 MB of memory for $m = 20$ (compared to 52 GB if we had allocated the full table).

3.4.3 Calculating Branch Entropy

After recording the information in the tables, we calculate branch entropy. Formula 3.4 is used to calculate the entropy of every OFT entry for every unique branch. Then we take a weighted average over all entropy values using Formula 3.5. This leads to two numbers, representing the branch entropy for the global and local history when using m history bits.

To get the branch entropy for a smaller number of history bits, we collapse the tables to reduce the history size. To get the table for $m-1$ history bits, the oldest history bit, represented by the most significant bit of the index, needs to be removed. After removing this bit, the two entries with the same index need to be merged to one entry. This is done by adding the taken and not-taken counters of the different entries (see Figure 3.6). The table now represents the OFT for $m-1$ history bits and can be used to calculate the new entropy number. This process can be repeated until the number of history bits equals 0 (e.g., to model a simple bimodal predictor). The result of this step is two sets of entropy numbers, one for local and one for global history, containing entropy numbers for history sizes from 0 to m.

Tournament predictors: These entropy numbers reflect two basic branch predictor organizations: a GAp predictor, which uses global history and branch address bits to index its PHT, and a PAp predictor, which uses local history and branch address bits. Tournament predictors [45] combine multiple predictors to obtain better prediction accuracy. They aim at selecting the optimal history (local or global) per individual branch. Therefore, next to global and local history entropy, we also calculate *tournament entropy*: we calculate global and local entropy per branch, take the minimum of the two, and average that minimum over all branches. This adds a third set of entropy values to the

application branch profile. Note that this does not change the way the tables are recorded; this only affects the entropy calculation step.

Warming: A PHT entry in a branch predictor contains meaningful information only after a few updates to that entry. So, even if a branch has stable behavior, the first few predictions can be wrong, depending on the initial value of the saturating counter. To account for this warming effect, we assign an entropy of 1 to the first access, which boils down to modeling a probability of 50% to predict the first branch correctly. This only has a noticable impact on the final entropy number if there are many different branches and branch histories that occur only a few times.

Aliasing: A branch predictor can also suffer from aliasing: because a branch predictor does not use the complete branch address, different branches can map to the same entry, which can lead to positive interference (all branches have the same outcome) or negative interference (the branches have different outcomes). To model this, we gradually reduce the number of address bits, and add all tables that have the same truncated address element by element, similar to collapsing the tables to model a smaller number of history bits. In the extreme, we use no address bits, meaning that all tables are added together. This models a predictor that uses no address bits, such as a GAg predictor. Modeling aliasing into the entropy adds multiple extra sets of entropy numbers, one for each setting of the number of address bits used to index the branch predictor PHT. Modeling aliasing also affects only the entropy calculation step, and has no impact on the way the tables are recorded. We will show the necessity of modeling branch aliasing in Section 3.5.5.

The resulting branch entropy profile for an application consists of multiple sets of entropy numbers: a set for local, one for global, and one for tournament history. Each set has an entropy number for each considered history length and number of address bits. Recording all of these values is necessary, as each application behaves differently when the history length and number of address bits is changed. Note that all of these numbers are calculated using a single profiling step. One can also choose not to model some effects, such as not taking into account warmup for predictors with adaptable history length (that combine the short warmup time for small histories and the accuracy of long histories, see Section 4.1), or not modeling aliasing for predictors that have aliasing-reducing hashing functions.

3.4.4 Model Construction

The last component involves the construction of the model that relates entropy values to branch misprediction rate. Because different branch predictors have a different misprediction rate for the same application, each type of branch predictor will have its own model. We build this model using the entropy numbers and the branch predictor misprediction rates for the benchmarks in the training set. Selecting the training set is straightforward but needs to be representative: because we have to measure the branch entropy profile for each

application, we can select a subset of applications that covers the full range of entropy values, leaving out applications that have a similar entropy to that of already selected applications.

We first select the set of entropy numbers that is conceptually close to the predictor we want to model: local, global or tournament history. For common predictors (e.g., bimodal predictor, GAg, GAp, PAp, etc.), this is straightforward. For other, more complex predictors (e.g., perceptron-based predictors), this might not be so obvious. For these predictors, we can build a model for each set of entropy numbers, and select the one that has the smallest error on the training set. Building the model only takes a fraction of time once the entropies and branch misprediction rates are known.

Because we have an entropy number for each number of history bits, we do not need to construct a new model for each history length for a particular type of branch predictor. Instead, we can fit the entropies and misprediction rates for all history sizes and for all benchmarks in the training set to a single model. This reduces the number of models, and also increases the number of points to fit the model (or it reduces the number of simulations required to produce enough training data). Furthermore, because we incorporate branch address aliasing in the entropy calculation, we can also incorporate the entropies and misprediction rates for a varying number of address bits for indexing into a single model.

We find that a linear relationship between entropy and misprediction rate fits best. This can be intuitively explained by our choice of using a linear entropy function: a branch predictor usually predicts the outcome most encountered in the same context in the past, so the fraction of the least encountered outcome correlates well with the misprediction rate. We use the following equation to calculate the misprediction rate M from the entropy E:

$$M(E) = \alpha + \beta \times E. \tag{3.6}$$

Parameters α and β are determined using a least squares fit.

Figures 3.7(a) and 3.7(b) shows the result of such a fit. Every point represents the branch predictors misprediction rate (Y-axis) and the corresponding entropy value (X-axis). There are 40 benchmarks and 20 different history lengths, resulting in 800 points. The fitted model is indicated by the dashed line. It is clear that a linear model is appropriate for this data, and that the fit is relatively good.

3.5 Results

We now evaluate the accuracy of the proposed branch predictor model.

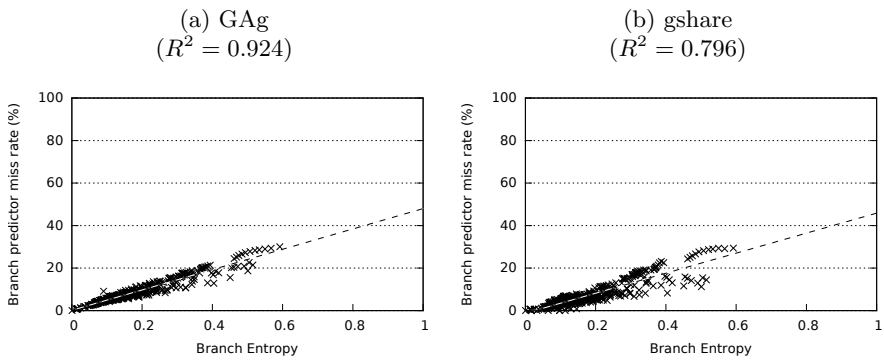


Figure 3.7: Graphical representation of the training input and the fitted model.

3.5.1 Experimental Setup

To construct and evaluate the proposed branch predictor model, we use the framework provided by the 2011 Championship Branch Predictor competition (CBP)[2]. This framework consists of a set of 40 benchmarks from different domains (client, server, work station, multimedia and integer) and a simulation environment in which we can easily implement different branch predictors. We implement our entropy profiling tool within this framework. We also evaluate our branch predictor model using the SPEC CPU 2006 benchmarks. Thereto, we implement our profiler and a branch predictor simulator in Pin [41]. We construct one-billion instruction simulation points using SimPoint [54] for all benchmarks with all reference inputs (leading to 55 benchmark-input combinations).

Recording the tables with the branch history patterns takes approximately the same time as simulating a branch predictor, so measuring entropy has the same overhead as one branch predictor simulation, but enables predicting misprediction rates for all modeled branch predictors. Using the model as a replacement for simulation therefore leads to a speedup equal to the number of evaluated branch predictor configurations. Note that the training phase needs to be done only once per predictor, on a limited set of applications and history lengths (e.g., 20 points per predictor to estimate the two parameters of the model should be sufficient if a good entropy coverage is present in the training set). We notice that some benchmarks – in particular the SPEC benchmarks – show significant phase behavior, leading to different behavior for the same branch at different points in time during the execution. We therefore profile the entropy for every interval of one million instructions, and calculate the final entropy number as the average entropy across all intervals, weighted by the number of branches per interval.

We also consider the most recent 2014 Championship Branch Predictor competition, but find that the provided benchmarks have a narrower entropy

[2]Available at http://www.jilp.org/jwac-2/

range than those from the 2011 competition. The benchmarks from the 2014 competition are within the $[0, 0.21]$ entropy range, while those from 2011 range within $[0, 0.39]$. Because our model is constructed using regression, a larger range will result in a more robust model. We therefore choose to use the 2011 benchmarks to evaluate the model (we could not simulate the 2014 benchmarks on the 2011 infrastructure and vice versa, preventing the evaluation of a superset consisting of the 2011 and 2014 benchmarks). Nevertheless, we ported the predictor code of the 2014 winning predictors to the 2011 framework to evaluate our model on the latest state-of-the art predictors (see Chapter 4).

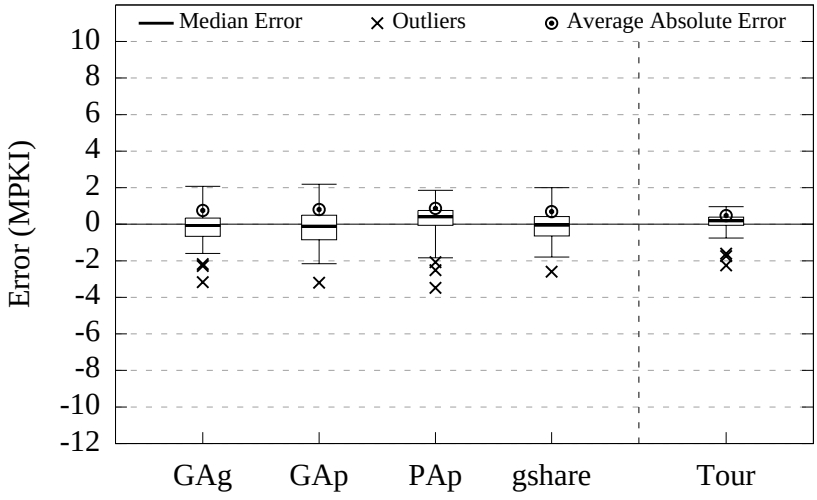
3.5.2 Model Accuracy

We first show that linear branch entropy correlates well with branch misprediction rate, by evaluating the accuracy of the branch predictor model. Because the model is a linear function, model accuracy is a good indicator for correlation: if the model is accurate, branch entropy correlates well with misprediction rate, and vice versa. We evaluate the accuracy of the model for a few common two-level predictors: a GAg predictor, a GAp predictor, a PAp predictor and a gshare predictor. Furthermore, we also evaluate a tournament predictor, consisting of a GAp and a PAp predictor, with a metapredictor (indexed with address bits only) to choose between the two predictors.

We evaluate the model using leave-one-out cross-validation: we train the model on all but one benchmark, and evaluate the accuracy for the left-out benchmark; and we repeat this process for all benchmarks as the left-out benchmark. We report the average difference in MPKI (misses per 1,000 instructions) between prediction and simulation. Using MPKI instead of misprediction rate avoids inflating numbers when there are few branches. Furthermore, MPKI is proportional to the branch miss CPI penalty of an application [19].

We train the model for each of the predictors using simulation results for all training benchmarks and for history sizes between 0 and 20 bits. For GAg, gshare and GAp, the global history entropy is used to fit the model to the misprediction rates; for the PAp and tournament predictors, we use local and tournament history entropy, respectively. For GAg, we calculate a single global entropy number across all branches (by combining all per-branch tables into one table), instead of an entropy number per branch. For gshare, we find that using per-branch entropy (as in the GAp predictor) leads to relatively large errors. By XOR-ing address bits and history, we loose some of the information of the address bits. We find that fitting the misprediction rates for gshare to the global entropy with a single entropy number across all branches (as for GAg) provides the best results. This can be explained by the fact that we use the same amount of history bits as for the GAg predictor to index the pattern history table (PHT), and that the XOR with the address bits partly solves the problem of GAg that the same history for different branches is mapped to the same entry (history aliasing). This aliasing reduction effect is now visible in a low (even negative) parameter α for the gshare model, which is a measure for aliasing, as we will discuss in the next section.

(a) Prediction error for the CBP 2011 benchmarks.



(b) Prediction error for the SPEC CPU 2006 benchmarks.

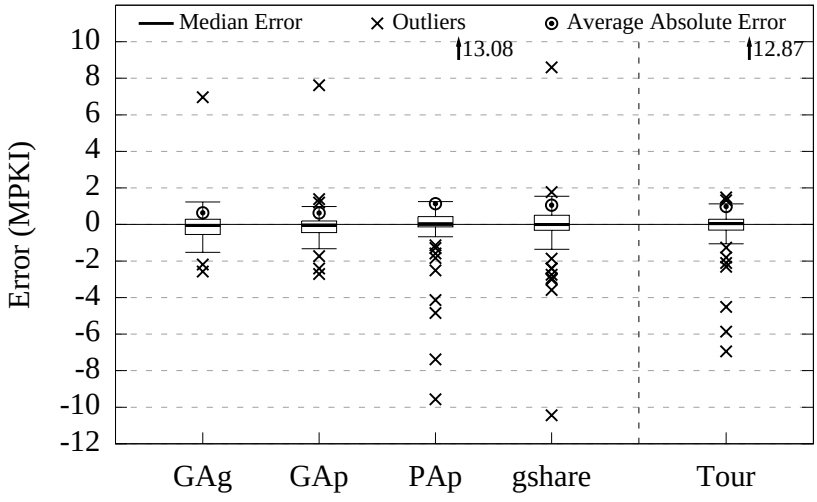


Figure 3.8: Prediction error as a boxplot, showing median and average absolute error.

The prediction error is shown as a box-and-whiskers plot[3], see Figures 3.8a and 3.8b, and per benchmark, see Figures 3.9a and 3.9b, for the five branch predictors, on CPB and SPEC, respectively. These numbers are for one specific configuration (i.e., number of history and address bits) for each predictor. These configurations have approximately the same hardware cost (4 K bytes at the second level). The error of the tournament predictor is the smallest for CBP with an average absolute error of 0.36 MPKI; for SPEC, the smallest error is observed for the GAp predictor with an average absolute error of 0.63 MPKI. For all modeled predictors, the average absolute error is around 0.70 MPKI and 0.89 MPKI for CBP and SPEC, respectively. The average MPKI for all predictors is 10.8, which means that the model has a relative error of less than 20%, and the errors are both negative and positive. The highest average errors are observed for the PAp predictor (0.87 MPKI and 1.14 MPKI for CBP and SPEC, respectively) and gshare (0.69 MPKI and 1.06 MPKI). This is because these predictors suffer from PHT aliasing, which, in contrast to pure branch address aliasing, is not modeled in our entropy calculation. The PAp predictor suffers from aliasing in its history table: we use 10 bits to index the history table, which means that it may happen that different branches update the same history, which pollutes this history. This effect is not modeled in our entropy calculation. Modeling this would require separate tables for every setting of the number of bits used to index the branch history table, which would incur too much overhead. The gshare predictor XORs history and address bits, which also leads to unpredictable aliasing effects (branches with a different history and instruction address may map to the same entry).

Although the models for CBP and SPEC have similar average errors, the SPEC benchmarks have more outliers. Some of the SPEC benchmarks show more irregular behavior than the CBP benchmarks. In particular, gcc has many unique branches, causing a lot of aliasing effects in the first level of the branch predictors. As a result, the branch misprediction rate is often predicted too low, leading to negative errors. The extreme points at the negative side in Figure 3.8b for the PAp, gshare and tournament predictor, are all for gcc with different input sets. There is also one outlier on the positive side, which is dealII for all predictors. We find that the branches in dealII show very fine-grained phase behavior: during a few thousand instructions, a particular branch is first taken multiple consecutive times, and then not-taken multiple times. Our entropy metric aggregates the outcomes per one million instructions, leading to a large entropy value for this branch (because it is about half of the time taken and half of the time not), but a predictor can accurately predict the sequences of equal outcomes, only missing when the outcome flips. Reducing the entropy measuring interval time would solve this problem, but incurs more overhead in the profiler, and also makes it impossible to detect long history patterns. Because we see this behavior in only one of the 95 evaluated benchmarks, we deem this additional overhead is not worth the gain in accuracy.

[3]The box covers the second and third quartile, with a line at the median. The whiskers cover all points within 1.5 interquartile distance outside of the box, and the crosses are the outliers. The circles represent the average of the absolute value of the error.

(a) Prediction error for the CBP 2011 benchmarks.

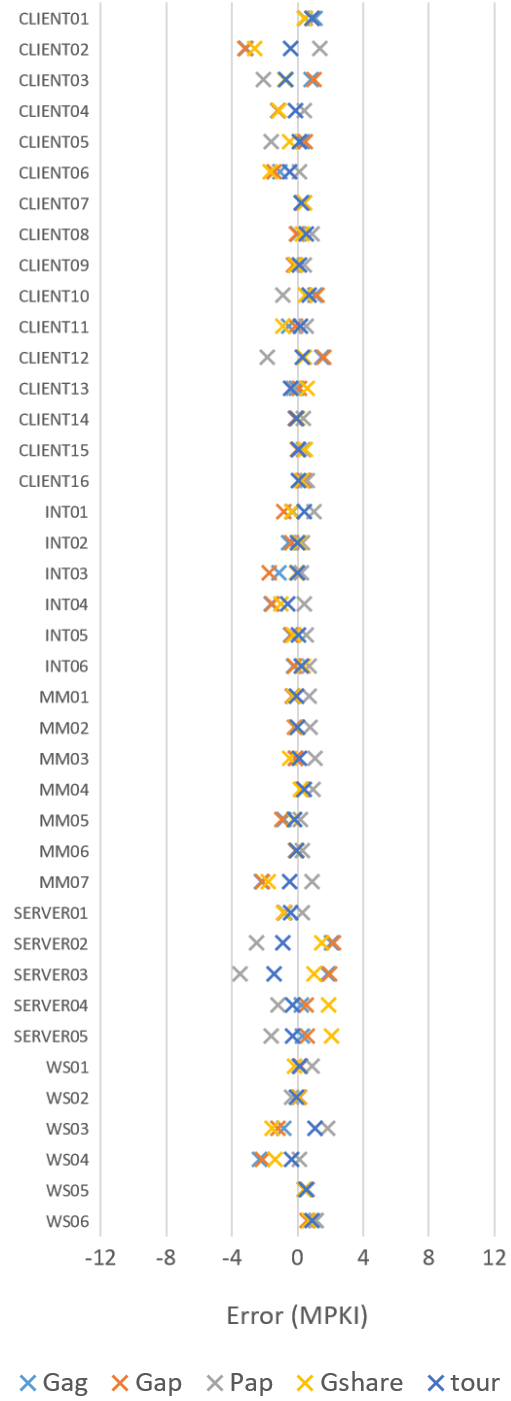


Figure 3.9: Prediction error in MPKI for the different predictors.

(b) Prediction error for the SPEC CPU 2006 benchmarks.

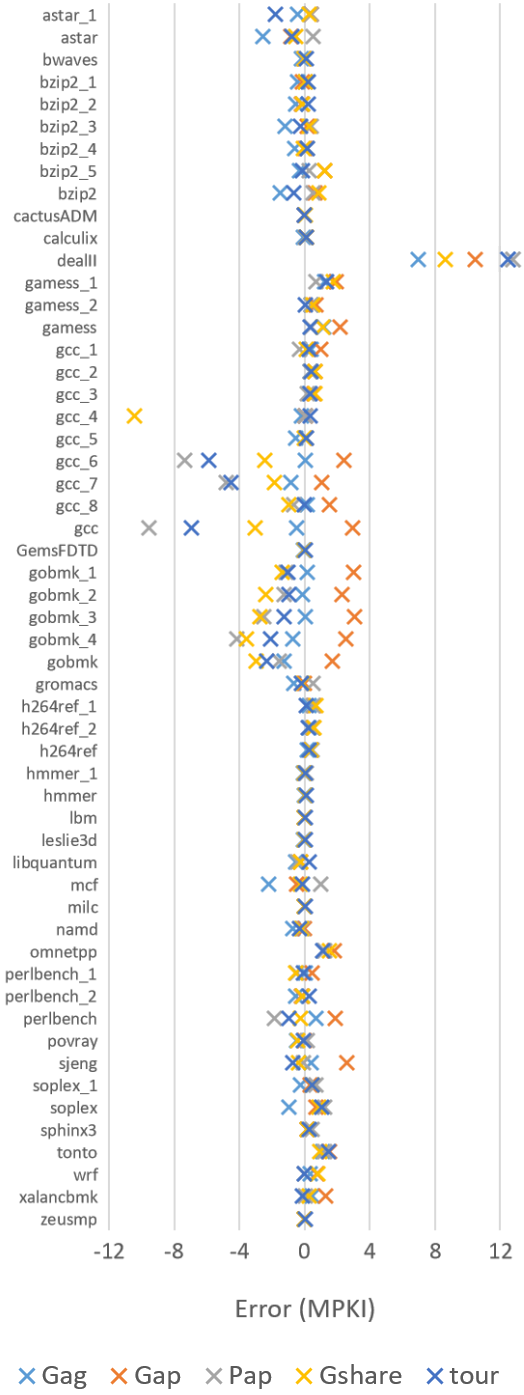


Figure 3.9: Prediction error in MPKI for the different predictors.

(a) CBP

	α	β
GAg	-1.047	51.323
GAp	-1.037	51.530
PAp	3.450	55.722
gshare	-1.246	55.358
Tournament	-0.455	56.010

(b) SPEC

	α	β
GAg	-0.305	46.344
GAp	-0.154	45.185
PAp	-0.281	58.156
gshare	-0.189	53.322
Tournament	0.140	52.522

Table 3.1: Model parameters for the different branch predictors (in % miss rate).

3.5.3 Interpreting the Model Parameters

Tables 3.1a and 3.1b show the fitted α and β parameters for each of the predictors (in % miss rate), for the CBP and SPEC benchmarks, respectively. Given the linear branch predictor model, one can give an intuitive meaning to these parameters. α is the misprediction rate for zero entropy, and determines the misprediction rate for low-entropy branches, while β is the slope of the line, and therefore is an indicator for the misprediction rate for high-entropy branches. Note that when α is negative, the model would predict negative misprediction rates for low entropy numbers. Obviously, this makes no sense, so we estimate a zero misprediction rate in such a case.

The α and β parameters allow for comparing branch predictors. For example, the GAp predictor is good at predicting low-entropy branches (low α, see Table 3.1a), but has difficulties with high-entropy branches (high β), whereas the PAp predictor predicts difficult branches more accurately (low β), but has a higher miss rate for low-entropy branches (high α), due to warmup effects (i.e., a static branch has to be executed multiple times until the history contains valid information).

A complementary explanation for the α parameter is that it quantifies the impact of the intrinsic misprediction rate of the predictor due to aliasing, i.e., α is the offset that is added to all misprediction rates. For example, in Table 3.1a, we can see that PAp has the largest α of all predictors. On the other hand, for some predictors, α is negative, e.g., for gshare, in which case we estimate a zero misprediction rate for low-entropy branches. These predictors almost perfectly predict low-entropy branches, because they alleviate aliasing issues.

Although the model parameters can differ significantly for the two benchmark suites (e.g., for the PAp predictor), the difference is not as big as it may seem: a model with a low α and a high β may be similar to a model with a high α and a low β, in the typical entropy range, i.e., between 0.1 and 0.2. A cross-validation experiment between the results for CBP and SPEC shows that these models may seem different but are in fact fairly similar. Figure 3.10 shows the average absolute error for the models constructed using the CBP and SPEC benchmarks, respectively, and then evaluated on the other benchmark suite. The error only slightly increases when the model of a different benchmark suite is used, showing the robustness of the technique. Using the CBP model on

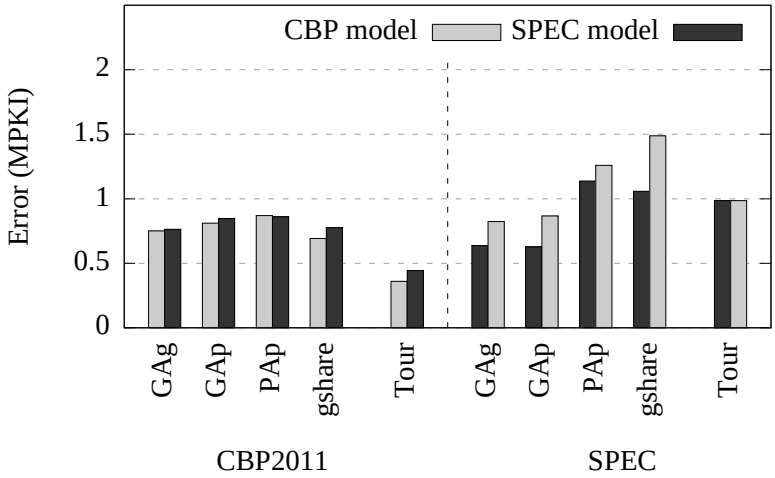


Figure 3.10: Average absolute error for the two models (constructed using CBP and SPEC benchmarks) on the two benchmark suites for common predictors.

the SPEC benchmarks increases the error slightly more than using the SPEC model on the CBP benchmarks; this is because the SPEC benchmarks show a uniform distribution of entropy numbers, whereas the CPB benchmarks show a more clustered and sparse distribution.

3.5.4 Comparison to Prior Work

Prior work proposed taken and transition rate for classifying branches (see Section 3.2). Although they do not specifically target estimating branch misprediction rates, they show that taken and transition rate correlate well with misprediction rate. Therefore, we also built models that use taken rate, transition rate and a combination of both. The models are built by dividing taken and transition rate numbers into bins, and using the average miss rate in a bin as the branch miss rate estimation (requiring a similar training phase as our technique). For the combination of taken and transition rate, we construct two-dimensional bins, combining the bins from taken and transition rate (for example, a bin contains all branches that have a taken rate between 0.1 and 0.2, *and* a transition rate between 0.6 and 0.7; this is similar to what is described in [24]).

Figure 3.11 shows the resulting error box plots for our example predictors and the CBP benchmarks. The figure shows that the combination of taken and transition rate is indeed better than taken or transition rate individually. It is also clear that the branch entropy method proposed in this work outperforms all three other methods. For the GAg predictor, the improvement is the highest: the average absolute error drops by more than 2× to 1.82 MPKI. For the other

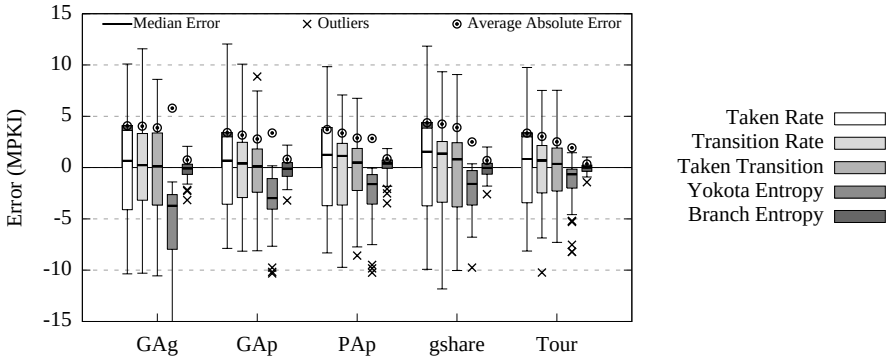


Figure 3.11: Model error in MPKI for taken rate, transition rate, the two combined, Yokota's lower bound, and linear branch entropy, for the CBP benchmarks.

predictors, the average absolute error drops from an average of 3.2 MPKI to 2 MPKI (38% reduction).

Other prior work by Yokota et al.[61] shows that branch entropy can provide an upper bound on the hit rate (or a lower bound on the miss rate), by reconstructing p (binary probability) from the entropy number. This means that there is only one model that is used for all branch predictors. Figure 3.11 also shows the error when we use only entropy (i.e., we divide linear entropy by 2 to reconstruct the p), and no specific fitted model per predictor. It shows that all errors are negative, i.e., the misprediction rate is always underestimated, which is consistent with the notion of a lower bound. The average absolute error for this method is 3.71 MPKI, which is higher than our model and which is also higher than the model using the combination of taken and transition rate.

Figure 3.12 shows the average MPKI estimation error for different predictor sizes (using different history sizes and numbers of address bits used to index the PHT), for both our model and the model using the combination of taken and transition rate (which is the most accurate of all prior proposals). As a reference, the average MPKI of the predictors is also shown. This graph shows that our model is more accurate than taken and transition rate across all sizes, and that the relative error remains approximately the same for small and large predictors.

3.5.5 Modeling Aliasing

Some branch predictors suffer from branch address aliasing: because only a few bits of the address are used to index the PHT, multiple branches may map to the same entry. We model this by combining the recorded outcome frequency tables (OFT) for branches that map to the same PHT entry (see Section 3.4.2). This has some impact on the complexity for calculating entropy, because we have to look for branch addresses that, when truncated, map to the same entry, and we have to add the tables element-wise.

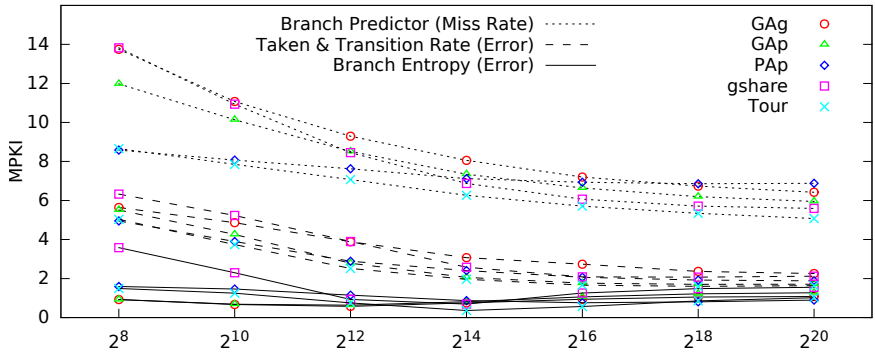


Figure 3.12: Average absolute MPKI error for different predictor sizes (the horizontal axis shows the number of entries in the second-level PHT) for the CBP benchmarks; as a reference, MPKI is shown as well for the various predictors.

Figure 3.13 shows the impact of (not) modeling aliasing for the GAp predictor. The top figure shows the model fit where no aliasing is incorporated in the entropy profile; the bottom figure includes modeling aliasing. It is clear that the fit is much better if we include aliasing. The average absolute MPKI error decreases from 1.22 to 0.64.

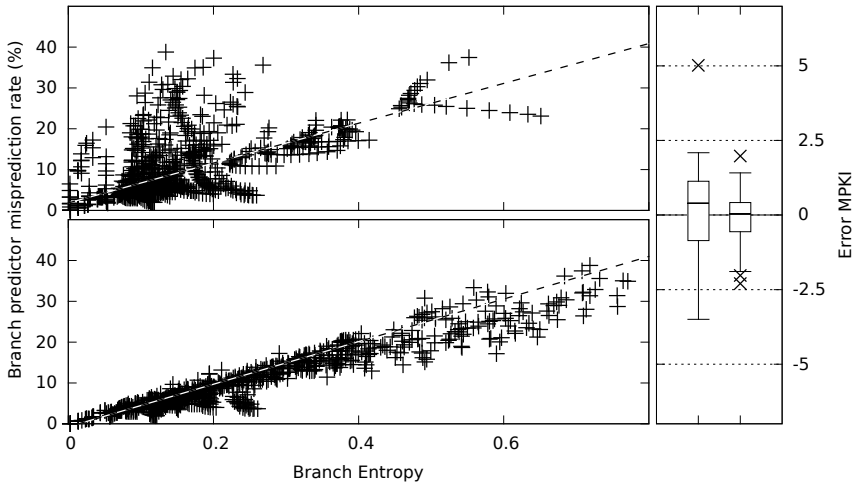


Figure 3.13: GAp model without taking aliasing into account (top graph) versus accounting for aliasing (bottom graph) for the CBP benchmarks. The error box on the left is without modeling aliasing, the right one includes aliasing.

Chapter 4

Branch Behavior Analysis

Now that we have shown that our model to estimate branch misprediction rates using a predictor-independent profile is accurate, the most straightforward way to use it is to predict misprediction rates for different branch predictors when performing a design space exploration. In the results section of the previous chapter, we have already shown that our model is accurate across a large design space. However, because we have a separate application profile (entropy) and branch predictor model (linear regression model), we can also use the model to compare branch predictors, which we will do in this chapter. We also show how entropy can be used to classify benchmarks and select a representative subset of benchmarks for branch predictor studies.

4.1 Comparing State-of-the-Art Predictors

As discussed in Section 3.5.3, the parameters α and β of the linear model can be used to compare branch predictors. As a case study, we now compare the top-four branch predictors from the 2014 Championship Branch Predictor competition in the 4 KB hardware budget category. These are, ranked by the number of mispredictions per thousand instructions (MPKI):

1. Seznec: TAGE-SC-L [51] (5.29 MPKI)
2. Otiv: H-Pattern [47] (5.47 MPKI)
3. Ishii: GL-TAGE [29] (5.58 MPKI)
4. Jiménez: Strided Sampling HPP [30] (5.81 MPKI)

Note that the MPKI values and the ordering differs slightly from the results presented at the CBP-4 workshop, because we evaluated these predictors on the 2011 CBP benchmarks. We will discuss the results using the 2014 benchmarks at the end of this section.

First, we need to select the entropy numbers that should be used to model the misprediction rate for these branch predictors. Since they all claim to

be able to handle large branch histories, we use 25 history bits, which is the largest history we used for measuring entropy[1]. Furthermore, we do not model branch address aliasing and warmup effects when calculating the entropy, assuming that these advanced predictors are able to eliminate these effects (e.g., by keeping multiple tables with different hashing and history lengths).

We also conjecture that tournament entropy should fit best, because all four use both global and local history. We verified this claim by constructing models using global history entropy, local history entropy and tournament entropy, and noticed that tournament entropy indeed leads to the lowest error on the training set.

The resulting average MPKI error equals 0.91 for Seznec's predictor, 1.14 for Otiv's predictor, 1.26 for Ishii's predictor and 1.39 for Jiménez' predictor (compared to an average MPKI of 5.54 for all four predictors).

	α	β
Seznec	-0.251	58.039
Otiv	0.063	56.238
Ishii	-0.035	58.629
Jimenez	0.782	51.016

Table 4.1: Model parameters for the state-of-the-art branch predictors (in %).

Table 4.1 shows the parameters of the linear model for the four predictors. Looking at the table we can see that the Seznec predictor has the lowest α, but also the second highest β. For the Jiménez predictor, it is the other way around: it has the highest α, but the lowest β. Figure 4.1 shows the models for the four predictors. Again, we assume a zero misprediction rate if the model estimates a negative misprediction rate. The figure shows that Seznec's misprediction rate is lower than for Jiménez if the entropy is lower than 0.147, but for larger entropies, Jiménez performs better, with a 2.47% lower misprediction rate if the entropy equals 0.5. The similarity between the models of the predictors of Seznec, Otiv and Ishii can be explained by the fact that they all extend upon the TAGE predictor, while Jiménez proposes a perceptron-based predictor.

	Entropy	Seznec	Otiv	Ishii	Jiménez
INT04	0.0022	0.09	0.10	0.10	0.63
INT06	0.0985	4.90	4.86	4.99	4.80
INT01	0.196	10.19	10.21	10.29	9.89
INT02	0.258	12.85	12.80	12.78	12.08
WS04	0.389	24.45	23.87	24.99	22.57

Table 4.2: Entropy and simulated misprediction rate (%) for the top-4 CBP predictors.

To ensure that this behavior is real and not an anomaly of our model, we show entropies and simulated misprediction rates for some of the benchmarks

[1] We see no significant decrease in entropy when we increase the history beyond 25 bits.

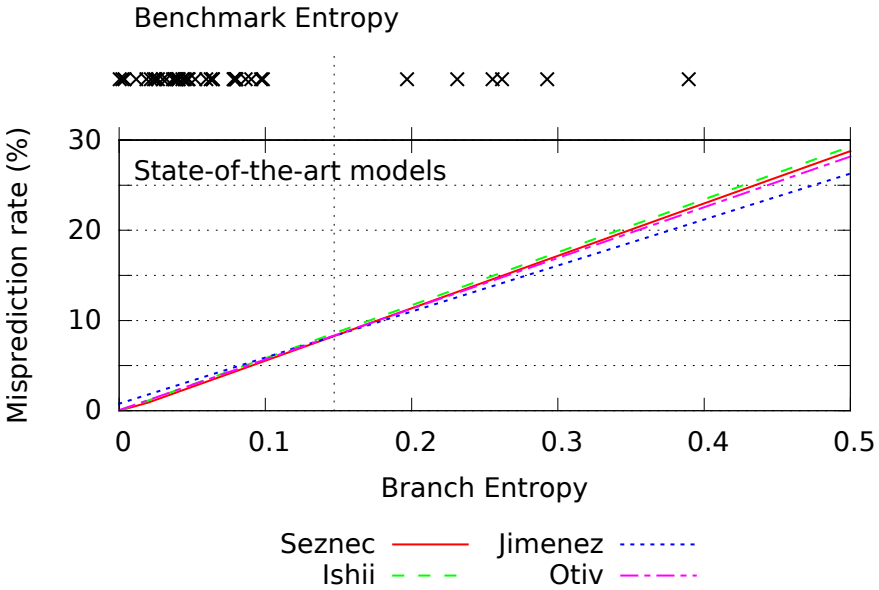


Figure 4.1: Models for the state-of-the-art predictors of the 2014 CBP championship. The entropies of all benchmarks are shown at the top.

in Table 4.2. We can see that for the benchmark INT04, which has a very low entropy, the Seznec predictor outperforms the Jiménez predictor. For benchmarks close to the cross-over point (INT06 and INT01), the misprediction rates for all predictors are very similar. For the high-entropy benchmark WS04, the Jiménez predictor has a 1.9% lower misprediction rate than the Seznec predictor. Of all benchmarks that have an entropy higher than the cross-over point at 0.147 entropy, the average misprediction rate of Jiménez is almost 1% lower than the misprediction rate of Seznec.

By looking at the entropy distribution of all benchmarks, shown on top of Figure 4.1, it is clear why Seznec's predictor performs best on average: only 6 out of 40 benchmarks have an entropy that is higher than 0.147 (the cross-over point is indicated by the dashed line). The average misprediction rate is therefore mostly determined by the low-entropy benchmarks, for which Seznec's predictor performs best. This is further exacerbated by the choice of benchmarks in the 2014 competition: only 3 out of 40 have an entropy higher than 0.147, with a maximum entropy of 0.21, which is only slightly higher than the cross-over point.

4.2 2016 Championship Branch Prediction

Our proposed branch entropy metric was used by the CBP organizers to provide a more diverse and balanced set of benchmarks for the 2016 edition [6]. To maximize branch behavior coverage, while limiting the number of bench-

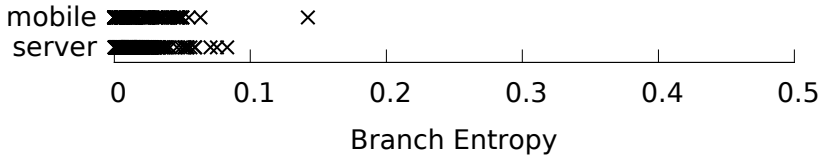


Figure 4.2: The entropy for all benchmarks used in the 2016 CBP.

marks in the benchmark suite, they constructed a branch behavior vector. This vector contains metrics like MPKI, IPC, ILP and misprediction rate for a set of predictors, static and dynamic instruction count, but also global, local and tournament entropy for a history size of 32 and 63 bits.

The result is a benchmark suite that consists of 440 benchmarks in two categories, server and mobile. To visualize the results, we use the tournament entropy, using 25 bits of history without warmup. This is the same configuration we used before in Section 4.1 to model the 2014 CBP winning predictors. Figure 4.2 is thus comparable with the top part of Figure 4.1.

It is clear from Figure 4.2 that all benchmarks have, on average, a very low entropy. In fact none of the 440 benchmarks has an entropy higher than 0.147 (the cross-over point from Section 4.1), whereas the 2011 CBP had 6 out of 40 benchmarks above an entropy value of 0.147 and 2014 CBP had 3 out of 40 benchmarks above 0.147. There is a clear trend towards low-entropy benchmarks and thus easy-to-predict benchmarks across subsequent editions of CBP.

4.3 Representative Benchmark Selection

In Section 3.5, we showed that our linear branch entropy metric correlates well with branch misprediction rate across a wide range of predictors. This means that benchmarks with a similar entropy also have a similar misprediction rate, and just produce redundant results in a study where branch predictors are compared. By only simulating benchmarks that have different entropy values, and therefore different inherent branch behavior, we can reduce the number of benchmarks that need to be evaluated.

To validate this, we cluster the 40 benchmarks in our setup based on entropy. Because we have multiple entropy numbers per benchmark (local, global, and tournament, and for different history sizes), we first perform principal component analysis (PCA) to reduce the number of dimensions. We find that 2 principal components (PC) already account for 98.3% of the variance, so we selected two dimensions. The result of PCA is illustrated in Figure 4.3. The first principal component (PC1) basically weights each entropy equally, and is therefore proportional to the average entropy across all history sizes. PC2 gives a negative weight to small-history entropies and a positive weight to large-history entropies: a PC2 close to 0 means that large and small history entropy are ap-

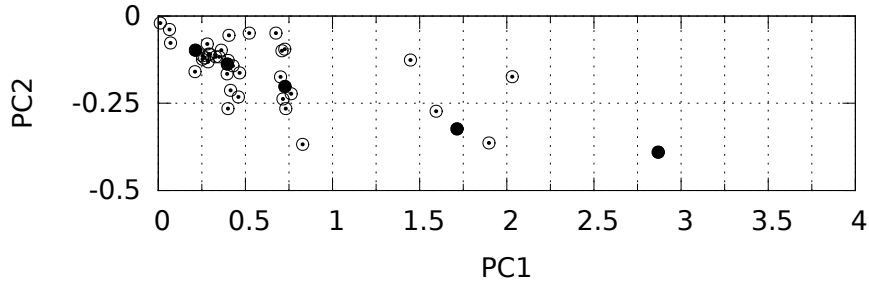


Figure 4.3: Result of PCA analysis and clustering for the 40 benchmarks. Cluster representatives are highlighted as a solid dot.

proximately equal, while a negative PC2 means that large-history entropy is lower than small-history entropy, so these benchmarks have more benefit when history is increased. PCA makes no particular difference between local, global and tournament entropy in its first two components.

We then cluster the benchmarks using these two principal components, by performing K-means clustering. We select an optimal number of clusters using BIC (Bayesian Information Criterion), which results in 5 clusters. For these 5 clusters, we select the benchmark that is closest to the cluster center as the representative benchmark; these are highlighted in Figure 4.3. We then simulate these 5 benchmarks on a branch predictor simulator, and weight the misprediction rates with the number of benchmarks in the respective cluster. This number is then compared to the average misprediction rate of the whole set of benchmarks. Table 4.3 shows the average misprediction rate across all benchmarks and the weighted average misprediction rate for the 5 representative benchmarks for different predictors. It also shows the average misprediction rate for a similar method using taken and transition rate as an input to the clustering mechanism (also for 5 clusters). For clustering using entropy, the difference between the actual and estimated miss rate is 4.5%, compared to 14.5% when using taken and transition rate. Furthermore, the clustering using entropy preserves the ordering of the branch competition top four, while using taken and transition rate classifies the Seznec predictor as the worst.

From this result we can conclude that linear branch entropy indeed gives a good representation of the predictability of the branches in an application. Therefore using entropy to group benchmarks with similar entropy numbers leads to an accurate clustering with a minimum number of clusters.

4.4 Impact of History Length

Sections 4.1 demonstrates that linear branch entropy a good representation is for the accuracy of state-of-the-art branch predictors, in spite of being based

Predictor	Miss rate	Cluster average using entropy	Cluster average using taken/trans. rate
GAp	14.69	14.62	15.71
PAp	10.26	9.58	11.04
GAg	8.54	8.29	9.82
gshare	7.12	7.56	8.27
Jiménez	4.95	5.46	5.57
Ishii	4.76	4.94	5.46
Otiv	4.66	4.93	5.54
Seznec	4.49	4.50	5.59

Table 4.3: Average misprediction rates (in %) across all benchmarks and average misprediction rate using entropy-based clustering and taken/transition rate based clustering. Predictors are ordered by decreasing average miss rate.

on the notion of how simple branch predictors operate. Modern branch predictors use much longer history lengths and are substantially more complex than the traditional gshare and two-level predictors. The question can be raised why linear branch entropy is also a good representation for more advanced branch predictors. We will do so by showing that branch entropy correlates well with branch predictability, even for state-of-the-art branch predictors with very long history lengths.

Prior work suggests that some branch patterns have a long period, and long history lengths are needed to capture these patterns [50]. But recording long history comes with a large hardware cost. Therefore modern predictors, such as the winning TAGE-based predictor from Seznec, will only record long history lengths for branches that benefit from it. Seznec's predictor uses 11 banks with different histories ranging from 0 to 359 bits. The predictor assigns easy-to-predict branches to a bank using little to no history, thus saving room to assign hard-to-predict branches to banks with (very) long history. For illustration purposes we select the first integer (INT01) benchmark from CBP. This benchmark has a linear branch entropy of 0.205, which is relatively high and will therefore contain branches that benefit from a long history.

Figure 4.4 shows branch entropy for two sets of branches for all history lengths of the 11 banks in Seznec's predictor, without taking warmup effects into account. The first set (dashed line) contains branches assigned to bank 3 (using 13 history bits); therefore this set represents branches that yield good accuracy using an average amount of history. The second set (solid line) contains branches from bank 10, using 359 history bits. These branches are classified as hard-to-predict by Seznec's predictor.

The difference between the two sets is noticeable over the whole range of history bits. When using no history information there is already a clear difference in branch entropy between the two sets, where the hard-to-predict branches start with an entropy of 0.63 and the medium set starts at 0.32. When increasing history size to 13 bits (history length of the third bank), there is a huge decrease in the entropy for both sets. But the entropy for the hard set

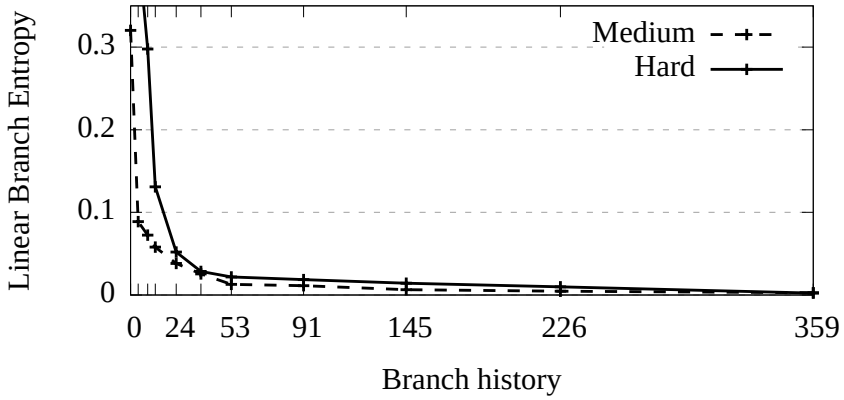


Figure 4.4: Impact of the number of history bits on medium and hard to predict branches.

is still 0.13, whereas the medium set already reaches 0.06. Although the entropy of the medium set still has a huge decrease in entropy up to 53 bits, these branches still end up in the third bank. This choice is probably due to size restrictions of banks with longer history. These higher banks are occupied by branches that benefit more from a longer history. In particular, the set of hard-to-predict branches have an entropy that is still twice as high when using 53 bits of history, and this behavior is constant when further increasing history length.

The key take-away from this analysis is that linear branch entropy correlates well with branch predictability, even for very long history lengths as observed in state-of-the-art branch predictors.

Chapter 5

Modeling Branch Penalty

Linear branch entropy was developed in the scope of a bigger effort to model the execution time of applications by only profiling microarchitecture-independent execution characteristics. This model was first published at the ISPASS 2015 conference [56]. To accurately estimate the complete execution time, we need a way to accurately estimate the penalty caused by mispredicted branches.

5.1 Branch Misprediction Penalty

As discussed in Chapter 2, the interval model forms the core idea of the single-threaded micro-architecture independent model. The interval model can be described by Equation 2.1. In this equation the second component is the one estimating the branch penalty. This part of the equation is replicated in Equation 5.1.

$$C_{branch} = m_{bpred} \times (c_{res} + c_{fe}) \tag{5.1}$$

Three values are needed to calculate this equation. The first value m_{bpred} is the number of mispredicted branches; the second value c_{res} is the branch resolution time; and the third value c_{fe} is the depth of the front-end pipeline. This last value is a fixed value and is set as a part of the design we want to analyze. We now describe how we estimate the number of mispredicted branches, and then how we estimate the branch resolution time.

5.1.1 Predicting the Number of Mispredicted Branches

The workflow is the same as described in Section 3.4 and is illustrated in Figure 5.1. The first step is to build the model for the branch predictor. To do so, a training run is needed. During this training run, a number of applications

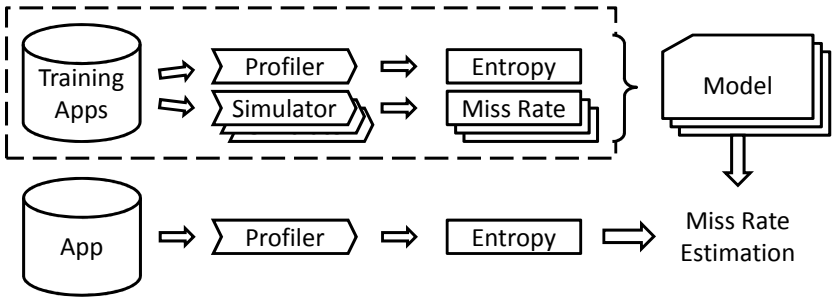


Figure 5.1: Schematic overview of how the branch predictor model works. The dashed box represents the training phase, which has to be done once for each branch predictor type.

are simulated with a branch predictor simulator to obtain the misprediction rates. In addition, the workload is profiled once to obtain the linear branch entropy. These numbers are combined to derive the model as explained in Section 3.4.

After building the model, the next step is to use this model to estimate the number of mispredicted branches for an unseen application. During the profiling phase, the branch profiling unit is gathering all branch instructions in order to build an entropy profile for the application. The branch profiling unit does not need to be sampled since it is fast enough, but the branch entropy information is stored for every window of 1 million instructions. During the integration into the single-threaded model this proved to increase the accuracy by improving the branch unit to catch any phase behavior an application may exhibit during its execution.

The branch predictor used during this study is the same tournament branch predictor as used to evaluate the results in Section 3.5. The model from Table 3.1b can be reused and is shown in Equation 5.2. During the modeling phase, the tournament entropy is used in the equation. The result is the branch predictor misprediction rate in percentage. If multiplied with the number of branch instructions, the result is an estimation for the number of mispredicted branches.

$$m_{bpred}(E_t) = (0.14 + 52.52 \times E_t) \times N_{branch} \tag{5.2}$$

5.1.2 Branch Resolution Time

The branch resolution time is defined as the time it takes for the branch to get executed after entering the reorder buffer. This depends on the length of the dependency chain leading up to the branch instruction. To calculate the branch resolution time, we use an algorithm, called the 'leaky-bucket' algorithm devised by Michaud et al. [46].

Algorithm 1 Leaky bucket model: calculating the branch resolution time.

```
1:  while N_i ⩾ D do
2:      if ROB_i + D ⩽ ROB then
3:          N_i = N_i − D;
4:          ROB_i = ROB_i + D;
5:      else
6:          N_i = N_i − (ROB − ROB_i);
7:          ROB_i = ROB;
8:      leave = min(I(ROB_i), D);
9:      ROB_i = ROB_i − leave;
10: c_res = lat × ABP(ROB_i);
```

The 'leaky-bucket' algorithm is shown as pseudo code in Algorithm 1. The inputs to the algorithm are the number of independent instructions as a function of the reorder buffer size ($I(ROB)$), the distance between two branch mispredictions (N_i), and the width of the front-end pipeline (D).

The first part (lines 1 to 9) of the Algorithm estimates the number of instructions that are in the ROB (ROB_i) by the time the second incorrect predicted branch gets dispatched. To do this, it simulates the dispatching (lines 2 to 7) and committing (lines 8 and 9) of instructions in the reorder buffer. When there is enough room in the reorder buffer to fit all instructions residing in the front-end pipeline, all D instructions enter the reorder buffer, otherwise the reorder buffer gets filled to capacity. N_i gets decreased with this number of instructions as the processor comes closer to dispatching the branch instruction. The number of instructions that leave the reorder buffer every cycle is estimated by the number of independent instructions that on average reside in a reorder buffer containing ROB_i instructions ($I(ROB_i)$) but can never by higher than the width of the pipeline (D).

When the branch instruction enters the reorder buffer, the branch resolution time is estimated as the average instruction latency (lat) multiplied by the average branch path[1] (ABP) for the number of instructions that are in the reorder buffer ($ABP(ROB_i)$).

5.2 Results

To evaluate linear branch entropy against simulation and to show that it outperforms the proposal of Yokota et al. [61], we implement both strategies in the profiler and modeling tool of the single-threaded model to estimate the number of mispredicted branches (m_{bpred}). Next, we calculate the branch resolution time as described in the previous section. Combining the predicted number of mispredicted branches with the predicted branch resolution time provides an estimate for the branch component.

[1]The average branch path is equal to the average number of producing instructions leading to a branch instruction.

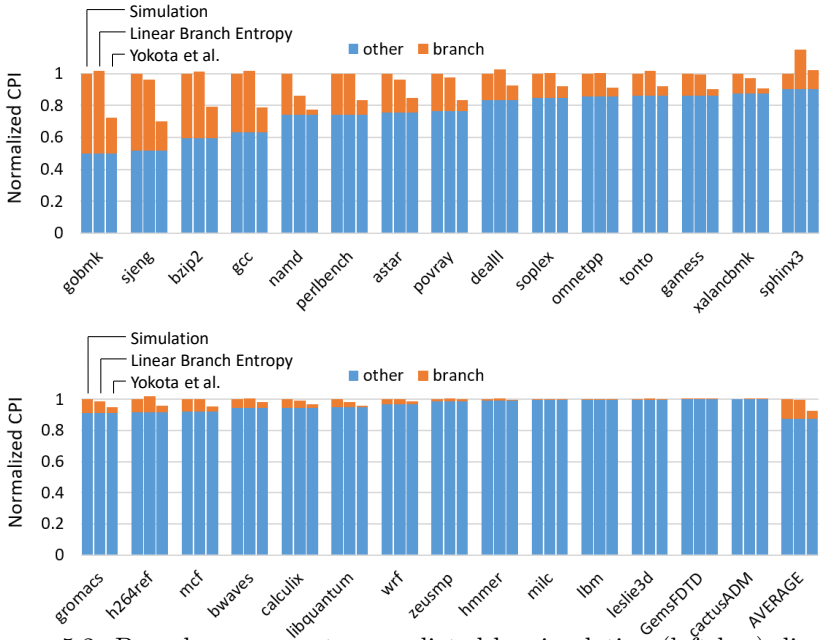


Figure 5.2: Branch component as predicted by simulation (left bar), linear branch entropy (middle bar) and Yokota et al. (right bar). The benchmarks are sorted by decreasing branch CPI component. The results are normalized to simulation.

The results are shown in Figure 5.2. The results are normalized to simulation which is shown as the left bar, the bar in the middle is the result from linear branch entropy, and the right bar are the results from Yokota et al. The bar is a representation of the complete execution time, where the highlighted part at the top represents the impact of the branch behavior.

The error of the branch component for the linear branch entropy model equals 1% on average, whereas Yokota's model, on average, has an underprediction of around 60%, leading to a 7% error on the complete execution time. This can be seen in the last stack where the weighted average of all benchmarks is shown. It is clear that the linear branch entropy yields an almost perfect prediction of the branch component whereas the Yokota et al. model estimates only half of the real branch impact.

The biggest error for our linear branch entropy method is observed for the namd and sphinx3 benchmarks. This is due to a poor fit of the model for these particular benchmarks. More specifically, the average entropy for sphinx3 equals 0.085, so when using the model from Equation 5.2, the resulting estimation of the misprediction rate is around 4.6% while the simulated misprediction rate is 2.21%. The entropy for namd equals 0.024, leading to an prediction of 1.40%, while the simulated misprediction rate is 3.47%. Although there are other benchmarks with a poor fit and thus an inaccurate estimation, the error for namd and sphinx3 is increased by the relatively large impact of the branch component.

Chapter 6

Guiding If-Conversion

In this chapter, we show how branch entropy can be used to guide the compiler for performing if-conversion. If-conversion is a well-known compiler technique that replaces branches by predicated execution [2]. Conceptually, it transforms control dependences into data dependences. If-conversion can result in a speedup or slowdown, depending on the balance between the benefits of removing the branch instruction, and the penalty of introducing additional data dependences. The compiler thus has to decide which branches to convert to obtain a net performance improvement.

Modern compilers (e.g., LLVM and the GNU C compiler `gcc`) use heuristics to steer if-conversion: they primarily consider the size of the basic blocks that are to be converted and the available instruction-level parallelism (ILP). Predicated instructions need to be fetched and decoded — and possibly executed if the instruction set supports partial predication as in x86 — even if the predicate turns out to be false. Therefore, if-conversion is only beneficial if the basic blocks are small, avoiding too much overhead for fetching (and possibly executing) useless instructions. If-conversion also introduces new data dependences, hence, if ILP is limited already, if-conversion may reduce it even more.

The basic premise of our proposal is to if-convert hard-to-predict branches. If branches are predicted well by the branch predictor, the control dependences are removed, so adding more data dependences is likely to hurt performance. On the other hand, if a branch is frequently mispredicted, we can avoid the misprediction penalty using if-conversion. However, a compiler is typically unaware of which branches are hard to predict in hardware at runtime.

We use branch entropy to differentiate easy- from hard-to-predict branches. Although it does not depend on a specific predictor, branch entropy correlates well with the misprediction rate across a wide range of branch predictors. Branch entropy is measured using a single profiling run. After the profiling run, the application is compiled using the branch information to steer if-conversion. Because branch entropy is independent of the micro-architecture and its branch

predictor, profiling needs to be done only once for each application of interest, and the if-converted binary can be deployed across a variety of machines, as we will demonstrate. This is in contrast to traditional feedback-directed optimization [10] for which a new profile needs to be collected for every branch predictor (and application).

6.1 Prior Work

Mahlke et al. [43] show that if-conversion can reduce the number of branch mispredictions by more than 50% on average. They if-convert unbiased branches, because the branch predictors they evaluate via simulation do not use any history. Our branch entropy metric takes into account branch history, and we evaluate our technique on real hardware, containing history-based branch predictors. Choi et al. [14] analyze the impact of if-conversion on the branch misprediction rate on an Intel Itanium processor. They show that the actual performance gains of if-conversion are typically around 2-3%, much lower than what was found in previous studies, which were based on simulation. Hazelwood and Conte [25] use dynamic compilation to if-convert branches with high misprediction rates, leading to a 2.6% performance improvement on average and up to 14% for one application (evaluated using simulation). We use static compilation, which implies that we do not have any dynamic branch misprediction information. Moreover, our technique is independent of the configuration of the branch predictor, resulting in a binary that performs well across microarchitectures.

6.2 If-Conversion on x86 using LLVM

The x86 instruction set only supports partial predication [44]. It features one predicated instruction, namely conditional move `cmov`, which moves or copies the content of one register to another register if a specific condition is met. The condition can be any condition that is allowed for conditional branches, e.g., `cmoveq` performs the move if a previous compare instruction has equal operands. If-conversion on x86 is implemented by executing the basic block unconditionally, and at the end add one or more conditional moves to copy the results into the new registers if the condition is met. In case the condition is not met, the results are ignored. This implies there are severe limitations for using if-conversion on x86. The basic block cannot contain stores or I/O output instructions, which cannot be undone if the condition is not met. By extension, it cannot contain system calls or calls to library functions.

Modern compilers, such as LLVM, use heuristics based on the analysis of the static code. In particular, LLVM tries to minimize the total number of additional instructions and dependences (`gcc` uses similar heuristics). First, the instruction-level parallelism (ILP) needs to be high enough, because if-conversion introduces additional data dependences. Then, if the ILP is high

Benchmark	Input	Convertable branches	Converted branches Default	Converted branches New
astar	BigLakes2048.cfg	76	19	31
bzip2	chicken.jpg 30	76	47	13
gcc	166.i	2781	708	181
gobmk	13x13.tst	520	180	127
h264ref	foreman_ref_encoder	979	261	122
hmmer	nph3.hmm swiss41	263	125	2
libquantum	1397 8	36	20	5
mcf	inp.in	15	9	5
omnetpp	omnetpp.ini	425	292	15
sjeng	ref.txt	198	76	44

Table 6.1: Evaluated benchmarks and inputs, along with the number of convertable branches and the number of branches converted by if-conversion in LLVM (default) versus our technique.

enough, the critical path can only grow with a specified number of instructions. The code path grows by the if-conversion itself (by inserting `cmov` instructions), and because both the 'if' and 'else' paths need to be executed. If both these conditions are met, the branch is if-converted.

We advocate using branch predictability information to steer if-conversion, instead of heuristics based on static code analysis. Branch predictability is quantified using our linear branch entropy metric: branches with a high entropy are converted, while branches with a low entropy are not. After analyzing the initial results of our technique, we discovered another case where if-conversion almost always leads to a performance improvement: there is only one conditional block that is either executed or skipped, *and* that block is executed most of the time (e.g., an 'if-then' clause without an 'else', and the condition of the 'if' is almost always true). To capture this type of branches, we compute the taken rate of branches in 'if-then' constructs, and we if-convert the branches if the taken rate is close to zero. We discuss how much branch entropy and taken rate contribute separately in Section 6.4.3.

6.3 Experimental Setup

We perform all experiments on an Intel Core i7-860 (Nehalem microarchitecture). We implement our if-conversion algorithm in LLVM v3.5.0, replacing the default heuristics. If-conversion is part of the `-O2` optimization level, so we compare the performance of our technique to that of `-O2`. We select all C/C++ integer benchmarks of the SPEC CPU2006 benchmark suite that we were able to compile on LLVM. The floating-point benchmarks have very regular branch behavior, which means that if-conversion has no considerable impact for these benchmarks. LLVM could not compile the FORTRAN benchmarks. Table 6.1 lists the benchmarks and the reference inputs we used. We measure the execution time of the program using performance counters.

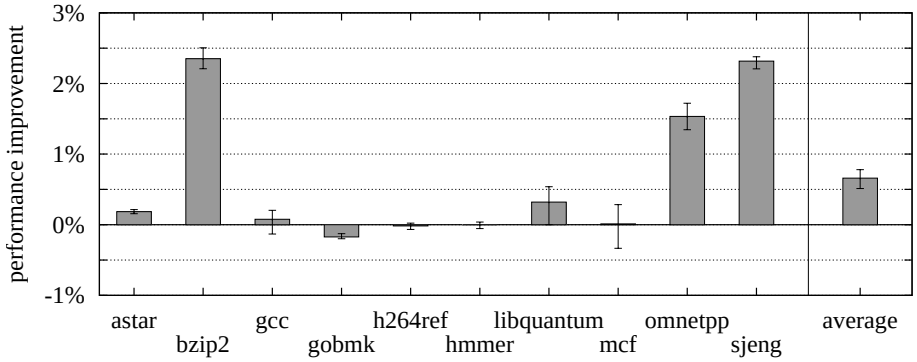


Figure 6.1: Performance improvement of our new if-conversion technique compared to the standard technique implemented in LLVM.

Currently, our profiler is implemented in the dynamic binary translation tool Pin [41], but is straightforward to be implemented in the standard profiling tool of LLVM. The profile is generated using the *training* input provided with the SPEC CPU2006 benchmarks, while the final performance is evaluated using the *reference* input. Each program is run 10 times, and we report the average performance improvement compared to standard `-O2`, along with the minimum and maximum observed improvement.

We decide to if-convert branches if the local entropy of a branch exceeds a certain threshold τ_e, or if a single-block branch has a taken rate that is less than τ_t. We use local entropy because this entropy number is not influenced by other branches in the application. So if we if-convert a branch based on its local entropy, this will not influence the entropy of the remaining branches. To find the optimal values for these parameters, we perform an exhaustive parameter sweep, and find $\tau_e = 0.18$ and $\tau_t = 0.16$ to be optimal. However, the results are not very sensitive to the specific values: the variability of the average performance improvement is less than 0.1% for any setting between 0.14 and 0.20 for τ_e and τ_t. We also do not convert branches that are infrequently executed (less than 5000 times) in the training input set. Branches that are not executed frequently will usually not have a big impact on performance, and they also provide too little information to obtain a stable and useful entropy number. Table 6.1 also lists the number of convertable branches (branches for basic blocks that do not contain stores, I/O, system calls, etc.), the number of branches converted by standard if-conversion and by our novel technique.

6.4 Results

Figure 6.1 shows the performance improvement of the benchmarks compiled with our new if-conversion criterion relative to default if-conversion in LLVM. We observe a distinctive improvement beyond default if-conversion for four of the benchmarks. The largest improvements are obtained for bzip2, omnetpp and sjeng, with improvements around 1.5% and up to 2.4% compared to de-

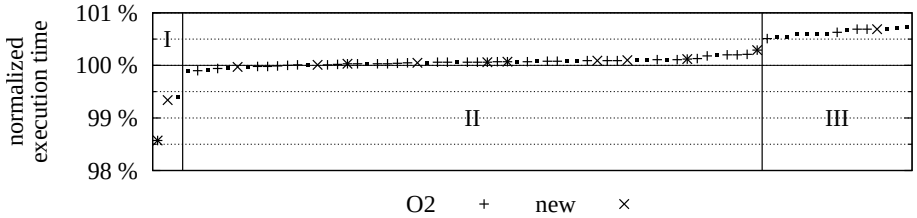


Figure 6.2: Analysis of the impact of every single convertable branch on the execution time for bzip2.

fault -O2. Our technique performs similarly to LLVM's if-conversion for the other benchmarks, with only gobmk suffering from a small 0.2% performance degradation, leading to a 0.66% average performance improvement across all the benchmarks. Interestingly, for all but one of the benchmarks, we if-convert fewer branches than standard if-conversion, see Table 6.1, while yielding better performance. This suggests we apply if-conversion more selectively and more effectively.

In the remainder of this section, we will analyze these results in more detail, starting by analyzing the converted branches for bzip2, the best performing benchmark, and then finding the reason for the poor performance of the gobmk benchmark. We will also discuss the contributions of entropy and taken rate separately, and we show results on other machines.

6.4.1 Analysis of bzip2

To obtain more insight into why we improve performance, we do the following experiment. We convert each of the convertable branches of bzip2 individually, and evaluate their impact on performance. Figure 6.2 shows the sorted normalized execution time (versus no if-conversion) for all convertable branches (76 in total). Some if-converted branches improve performance (zone I), while others degrade performance (zone III). The majority of branches (zone II) have a negligible impact on performance.

The figure also indicates the branches that our technique converts with an '×', and the branches that standard if-conversion converts are shown with a '+', (so branches with an '✳' are converted by both techniques). Compared to standard if-conversion, our technique converts fewer zone-III branches, which degrades performance, and more zone-I branches, which improves performance. However, we miss one of the three branches of zone-I, and we convert one branch of zone-III.

To explain the results, Table 6.2 lists more information about the three branches in zone I (branches 1 to 3) and the one branch in zone III that we convert (branch 4). The second column lists the execution time when that branch is if-converted (and only that branch), normalized to no if-conversion; this is the number shown in Figure 6.2. The next three columns contain profile information for the reference run without if-conversion: the number of times this branch is executed, its entropy and its taken rate. The three columns on

	Norm. exec. time	Profile info # Dyn [10^6]	Ent	TaR	Perf. counter change Instrs [10^6]	Branches [10^6]	Misses [10^6]
1	98.57 %	581.39	0.442	0.319	- 397	- 582	- 90
2	99.34 %	0.04	0.235	0.581	- 958	+ 64	- 107
3	99.39 %	11.40	0.030	0.985	- 865	- 12	+ 128
4	100.69 %	21.28	0.975	0.499	0	- 1	+ 51

Table 6.2: Profile and hardware performance counter information for interesting branches in bzip2.

the right contain performance counter data, showing the difference between no if-conversion and converting that branch in terms of the number of committed instructions, the number of committed branches, and the number of branch mispredictions.

The first two branches have a higher entropy than the threshold and are thus converted by our technique. The first branch behaves as expected: there are 582 million fewer branches, which is about the number of times this branch is executed, and there are 90 million fewer branch misses. There are also fewer instructions, which can be explained by the fact that for example a move and a branch instruction can be converted to a single conditional move instruction.

The second branch behaves differently. Although it is only executed 38 thousand times, converting this branch *reduces* the number of instructions by 958 million, *increases* the number of branches by 64 million, and *decreases* the number of branch mispredictions by 107 million. Obviously, this effect cannot be explained by the if-conversion of this branch solely. An analysis of the code shows that due to the if-conversion, other optimizations are triggered, with secondary effects resulting in a performance improvement.

We observe different results for the third and fourth branch. The third branch has an entropy of 0.03 and a taken rate of 0.99, which suggests that this branch is highly predictable. If-converting this branch would normally lead to performance degradation, because we then need to execute the code that the branch always jumps over. However, we notice a performance improvement. The number of branches decreases with 12 million, which is about the number of times this branch is executed, but the number of instructions also decreases considerably, which is unexpected. The number of branch mispredictions increases, but apparently, this extra penalty is undone by the decrease in the number of instructions due to the same code optimization of last branch. These two branches are part of the same code and converting one of these branches triggers another compiler optimization. If we would if-convert both branches, they do not reinforce each other, on the contrary, if-converting the third branch now introduces a performance degradation, as predicted by the low entropy and high taken rate.

Lastly, we consider the fourth branch in Table 6.2, which has a high entropy, and a 0.5 taken rate, suggesting that it is highly unpredictable. This seems like a good candidate for if-conversion, but if-converting this branch leads to

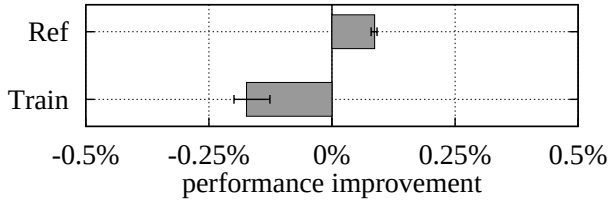


Figure 6.3: Performance improvement for gobmk relative to standard if-conversion (-O2) for our technique using the reference input (Ref) versus the training input (Train).

a performance degradation. By looking at the performance counter numbers, we again see unexpected behavior: the number of branches decreases by 1 million only, while this branch is executed 21 million times; this implies that new branches were added. The number of mispredictions increases unexpectedly, suggesting that the newly introduced (and possibly other) branches get mispredicted.

In conclusion, our premise that frequently executed high-entropy branches should be if-converted holds. If-converting less frequently executed branches can sometimes result in unpredictable behavior because of secondary effects, i.e., the code around the branch is rearranged, which causes a larger (positive or negative) performance impact than the impact of if-conversion. Nevertheless, our technique is still superior to standard if-conversion, by identifying frequently executed, hard-to-predict branches, and not if-converting easily predictable branches.

6.4.2 Analysis of gobmk

The performance benefits of if-conversion are limited in general — only few benchmarks benefit — yet we do not want to degrade performance beyond standard if-conversion for the benchmarks that do not benefit. We succeed in this, except for gobmk, which shows a small performance degradation of 0.2%. We find the reason for this slowdown to be a mismatch between the profiling run and the reference run. We obtained this insight by using the *reference* input, instead of the *training* input to collect the profiling information. We observed the result as shown in Figure 6.1 for all benchmarks, except for gobmk which is shown in Figure 6.3. Using the *reference* input for profiling leads to a small performance improvement compared to standard if-conversion. We conclude that a more representative training input is likely to solve the performance degradation of gobmk.

6.4.3 Branch Entropy versus Taken Rate

Our if-conversion criterion uses two criteria to decide whether a branch should be if-converted or not: entropy, which selects hard-to-predict branches, and taken rate, which selects almost-never-taken branches. In this section, we

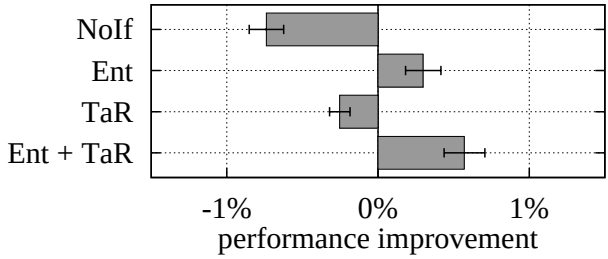


Figure 6.4: Average performance improvement relative to standard if-conversion for no if-conversion (NoIf), using only entropy as a criterion (Ent), only taken rate (TaR), and both (Ent + TaR).

show the contribution of both metrics separately. Figure 6.4 shows the performance improvement if we only take branch entropy into account for deciding whether to if-convert a branch ('Ent'); if we only if-convert 'if-then' branches with a near zero taken rate ('TaR'); and the combination of both ('Ent + TaR', which is our proposed technique). As a reference, we also show the performance of not if-converting any branch ('NoIf'). Again, all the numbers are relative to standard if-conversion (-O2). Only taking into account taken rate yields an average execution time that is slightly worse than standard if-conversion, but better than not converting any branch. When using entropy only, there is already a significant performance improvement over the LLVM's if-conversion. Combining both metrics, i.e., our proposed technique, outperforms using the individual metrics.

6.4.4 Impact across Microarchitectures

To check whether our results are consistent across machines, we evaluate the same binaries on two other processor architectures: an Intel Core i7 with a Sandy Bridge processor, and an Intel Atom with a Diamondville architecture (2-wide in-order pipeline). On the Sandy Bridge machine, the performance gains per benchmark are almost exactly the same as on the Nehalem machine. Figure 6.5 shows the results for the Atom processor. We observe similar results as for the Nehalem machine, however, now our top-3 performing benchmarks only get an average performance improvement around 1%. The main reason is the higher execution time on the Atom machine compared to the Core i7: more than 5× on average and close to 8× for mcf. In contrast, the number of mispredicted branches increases by 1.5× only (1.2× for mcf). So the impact of branch behavior on the total execution time is smaller than on the Nehalem machine, hence the smaller performance impact of our if-conversion algorithm. Note that the absolute performance improvement (decrease in execution time) of our technique on the Atom machine is larger than for the Core i7 machines, but due to the large execution time, the relative increase is smaller.

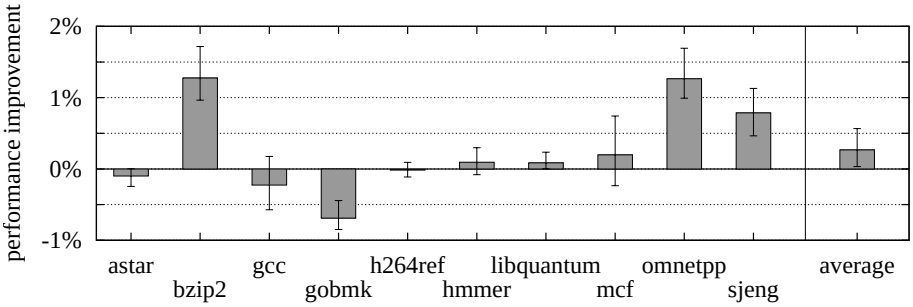


Figure 6.5: Performance improvement of our if-conversion algorithm on an in-order Intel Atom processor relative to default if-conversion.

Part II

Modeling Multi-Threaded Application Performance

The second part of this dissertation describes the model we developed to make rapid performance predictions of the execution time of multi-threaded applications on multicore hardware.

Chapter 7

Modeling Multi-Threaded Performance

7.1 Introduction

Simulation is the predominant methodology for computer architects to evaluate new processor architectures. Unfortunately, simulation is extremely time-consuming and tedious, especially when simulating multi-threaded workloads on multicore hardware. Analytical performance modeling is an attractive complement to detailed cycle-level simulation to quickly explore large design spaces at early stages of the design process. Several research groups have proposed analytical performance models for superscalar processors. These techniques can be broadly classified into three main categories: (1) empirical models, (2) mechanistic models and (3) hybrid models. Empirical models use training data obtained through simulation to create black-box models using machine learning and regression techniques, see for example [28, 37, 38, 39, 40, 48]. Mechanistic models are white-box models that capture the first-order mechanics of a processor, see for example [20, 58]. Hybrid models cover the middle ground through parameter fitting of a parameterized semi-mechanistic model, see for example [13, 23]. Empirical models are typically very accurate but do not provide insight and require extensive offline training. Mechanistic models are challenging to develop but once developed, they provide deep insight and do not require further offline training. This work seeks to advance the state of the art in mechanistic modeling.

Prior work in mechanistic modeling is limited to single-threaded processors. Interval modeling, developed over a series of research papers by Michaud et al. [46], then Karkhanis and Smith [34] and finally Eyerman et al. [20], models a superscalar processor performance by building up a CPI stack of components that represent useful computation versus lost cycles due to miss events. To collect the number of miss events (cache misses, branch misprediction rates, etc.), interval modeling relies on offline functional cache and branch predic-

tor simulation. More recently, Van den Steen et al. [58] improved upon this prior work by collecting only microarchitecture-independent characteristics of an application. The key advantage of doing so is that it allows for profiling the workload only once after which performance can be predicted for a range of previously unseen architectures. This prior work unfortunately is limited to single-core processors.

Mechanistic modeling of multi-threaded application performance is fundamentally more difficult than predicting single-thread performance. Not only do we need to accurately model per-thread performance, we also need to accurately model synchronization, resource interference and cache coherence effects. Moreover, as demonstrated later in this thesis, multi-threaded application performance prediction is further complicated by the fact that small prediction inaccuracies of the execution time in-between synchronization events, lead to an accumulation of errors across the entire program execution because application progress is determined by the slowest (most critical) thread.

Straightforward extensions of prior work towards multi-threaded workloads further motivate this work. Predicting multi-threaded application performance based on only the main thread or only the critical thread leads to an average performance prediction error compared to detailed simulation of 45% and 28%, respectively, and a maximum error above 110%. There are three reasons for the poor accuracy: (1) it does not model contention in shared resources, (2) it does not model cache coherence effects, and (3) it does not model synchronization overhead.

Some prior work focused on multicore performance prediction. Jongerius et al. [32] propose a multicore performance model for multiprogram workloads only. They focus on resource contention (i.e., negative interference) and do not model positive interference, cache coherence nor synchronization. Popov et al. [49] predict multi-threaded application performance using Amdahl's Law supplemented with the simulation of snippets of representative parallel code regions.

We propose RPPM (Rapid Performance Prediction for Multi-threaded applications) for predicting multi-threaded application performance on multicore hardware. A profiler collects a set of characteristics that captures a workload's execution behavior in a microarchitecture-independent way. The profile contains per-thread characteristics, as for the single-threaded model, as well as characteristics that affect inter-thread interactions, including shared memory access behavior and synchronization. The profile is then used to predict performance on a previously unseen multicore architecture. We assume that the number of threads considered during profiling equals the number of cores of the target architecture. A key feature of RPPM is that the profile needs to be collected only once, using which the performance can be predicted for a range of multicore architectures. Although the profile is measured during a particular multi-threaded execution, and therefore it may be subject to a particular inter-thread interleaving, we find it to enable accurate performance prediction across architectures.

We evaluate the accuracy of RPPM against detailed cycle-level simulation for all the OpenMP multi-threaded Rodinia benchmarks and a subset of the pthread-based Parsec benchmarks. RPPM predicts performance within 11.2% on average (23% max error) for a quad-core processor. We demonstrate the usefulness and applicability of RPPM for design space exploration studies and application performance analysis. In particular, we use RPPM to quickly identify the optimum among five design points with varying characteristics in terms of pipeline width and clock frequency while delivering the same peak performance (in operations per second). In addition, we use RPPM to construct bottlegraphs to analyze an application's parallel execution (im)balance.

7.2 Background

Modeling multi-threaded workload performance is challenging for at least three reasons. (1) One needs to accurately model per-thread performance. (2) One needs to accurately model inter-thread synchronization and interaction, including resource interference and cache coherence effects. (3) Because threads synchronize in a multi-threaded workload, there is an effect of accumulating errors. The latter is probably less well-known, hence we describe it next.

7.2.1 Accumulating Errors

In contrast to single-threaded performance modeling where performance prediction errors over relatively small execution regions are averaged out across the entire program execution, this is not the case for multi-threaded applications. Modeling multi-threaded performance is complicated by the fact that accurate predictions are needed in-between synchronization events, i.e., inaccurately predicting performance for the critical thread between synchronization events leads to an accumulation of error when predicting overall application performance.

We substantiate this claim through the following discussion. Without loss of generality, consider a barrier-synchronized multi-threaded application. (Other synchronization mechanisms such as critical sections and producer-consumer interactions face similar issues.) Predicting the execution time for each thread in an inter-barrier region may lead to over- and under-estimations for different threads. On average, we assume (expect) the average per-thread execution time to be predicted accurately for each inter-barrier region, i.e., the execution time may be over-estimated for some threads and under-estimated for others. However, even though the execution time predictions are accurate on average across all threads, this is not enough for multi-threaded workloads because the execution time of the inter-barrier region is determined by the slowest thread. Over-estimations of the execution time of inter-barrier regions lead to an accumulation of errors.

#Threads	Inter-barrier error		
	1%	5%	10%
1	0.00%	0.00%	0.00%
2	0.33%	1.67%	3.34%
4	0.60%	3.00%	6.01%
8	0.78%	3.89%	7.79%
16	0.88%	4.41%	8.83%

Table 7.1: Accumulating prediction errors in barrier-synchronized applications: overall prediction error as a function of thread count and inter-barrier prediction error. *Prediction errors accumulate because of synchronization.*

We illustrate this further using a micro-benchmark consisting of a loop of one million iterations for which each iteration takes the same amount of time. Assume now that the analytical model is 100% accurate on average but introduces some (random) over- or under-estimations within a given bound. The loop is parallelized such that n iterations run in parallel, with n the number of threads. All threads synchronize at a barrier after they have each executed one iteration. We run this micro-benchmark with different number of threads and different assumed inter-barrier prediction errors. The results are shown in Table 7.1.

When only a single thread is running, the over- and underestimations balance each other out and the resulting prediction error equals zero, i.e., we perfectly predict the average inter-barrier execution time, as expected. However, when running multiple threads, the execution time of an inter-barrier region is determined by the slowest thread reaching the barrier. As a result, the prediction error accumulates across barriers, leading to significant inaccuracies for predicting overall application execution time. We note that the error increases with increasing thread count.

7.2.2 Single-Threaded Performance Model

With this mind, we now provide a brief background on microarchitecture-independent analytical performance modeling for single-threaded applications, which we build upon to model per-thread performance in RPPM; we refer the reader to [58] for a more elaborate exposition of the single-threaded performance model. We next describe naive extensions to this prior work to predict multi-threaded application performance, which, as we will show in the evaluation, are inaccurate.

The single-threaded performance model consists of two steps. In the profiling step, we use a Pin tool [41] to collect an application profile containing only microarchitecture-independent statistics, i.e., these statistics are inherent to the workload and are independent of the underlying microarchitecture. In the prediction step, these statistics are used as input to the analytical model to predict the execution time on a particular processor configuration. Execution

time for a single thread running on an out-of-order processor is predicted using the following equation:

$$C = \overbrace{\frac{N}{D_{eff}}}^{\textbf{Base}} + \underbrace{m_{bpred} \times (c_{res} + c_{fr})}_{\textbf{Branch}} + \overbrace{\sum_{level=i} m_{ILi} \times c_{Li+1}}^{\textbf{I-cache}} + \underbrace{\frac{m_{LLC} \times c_{mem}}{MLP}}_{\textbf{D-cache}} \quad (7.1)$$

We distinguish four components in the model:

Instruction-level parallelism: The base component is obtained by dividing the number of micro-ops (N) by the effective dispatch rate (D_{eff}). The effective dispatch rate is a function of the width of the front-end pipeline, the available ILP in the application, and the amount of contention in the functional units. To accurately model ILP, we need fine-grained profile information, i.e., we collect statistics regarding instruction mix and inter-instruction dependences at the granularity of a thousand instructions, which we call a micro-trace. In order not to slow down profiling too much, we profile a micro-trace of a thousand instructions once every one million instructions. This allows us to characterize time-varying behavior in ILP at a moderate profiling cost.

Branch misprediction: The branch component quantifies the lost cycles due to branch mispredictions, as previously explained in great detail in Chapter 5. In short, the branch component is computed as the number of mispredictions (m_{bpred}) times the branch resolution time (c_{res}) (this is the time between the branch being dispatched into the issue queue and reorder buffer and its execution) plus the front-end pipeline refill time (c_{fr}). The number of mispredictions is calculated using the linear branch entropy model explained in Part I of this thesis.

Instruction cache: The I-cache component quantifies the impact of instruction cache misses and is computed as the product of the cache miss rate at each level (m_{ILi}) and the respective miss latency (c_{Li+1}). The cache miss rates are predicted using micro-architecture-independent reuse distance distributions using StatStack [18].

Long-latency loads: The D-cache component quantifies the time the core stalls waiting for main memory requests to resolve as a result of long-latency load misses. This component is computed as the number of last-level cache misses due to load instructions (m_{LLC}) times the average memory access latency (c_{mem}), divided by the amount of memory-level parallelism (MLP) or the average number of outstanding long-latency load misses if at least one is outstanding. MLP is computed using a microarchitecture-independent model as described in [57].

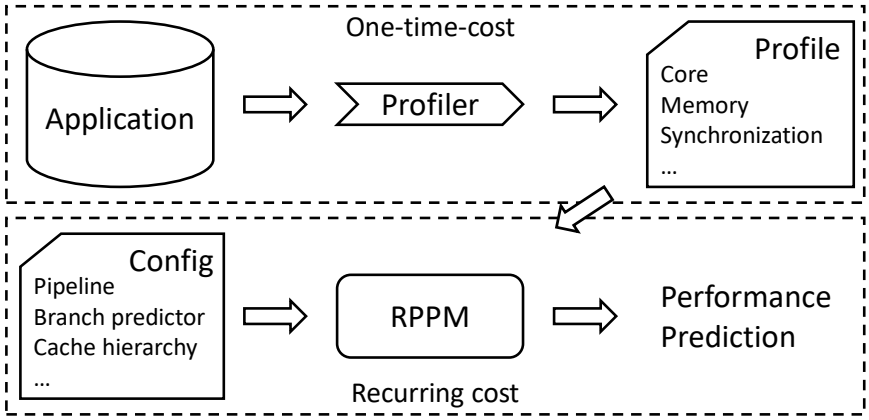


Figure 7.1: Schematic overview of the RPPM tool.

7.2.3 Naive Extensions for Multi-threaded Applications

We now discuss two naive extensions of the single-threaded performance model to predict the execution time of multi-threaded applications running on a multicore processor. In the evaluation, we will compare RPPM's accuracy against these approaches.

MAIN: In the first approach, we only profile the main thread. We define the main thread as the thread that gets initiated upon program execution; this thread completes the initialization phase before creating the other worker threads, and finalizes the execution once the worker threads have finished their execution. We apply the single-threaded model as described above to predict the execution time of the main thread. The predicted execution time for the main thread is then used as a prediction for the overall execution time of the multi-threaded application.

CRIT: The second approach profiles all application threads separately instead of only the main thread. After using the model to predict the execution time for every thread, the thread with the longest execution time will be marked as the critical thread. We then use the predicted execution time of the critical thread as a prediction for the overall execution time of the multi-threaded application.

Both these naive extensions do not properly take synchronization into account. Nor do they account for interference in shared resources and cache coherence effects. RPPM models both synchronization and shared resource interference, as we describe next.

7.3 RPPM

RPPM predicts multi-threaded application performance using two key components, see also Figure 7.1:

1. A profiler that collects microarchitecture-independent statistics including per-thread core characteristics, shared memory access behavior and synchronization events.

2. A prediction tool that takes these statistics as input and predicts multi-threaded execution time on a particular multicore processor architecture.

A key property of RPPM is that the profile contains a collection of characteristics that are independent of the underlying microarchitecture. Hence, we need to collect it only once and we can then predict performance for a range of microarchitectures. We note though that RPPM assumes the same number of threads during profiling as there are cores in the processor architecture for which we make the prediction. However, a single profile can be used to predict performance for a wide range of multicore architectures while varying clock frequency, pipeline width and depth, window and buffer sizes, cache sizes, cache hierarchies, branch predictors, etc. The fact that the profile is independent of the underlying microarchitecture speeds up design space exploration studies substantially.

In the next section we explain the workflow of RPPM in more detail. In the two sections following the next section, we are going into more detail of what the main differences are compared to the single-threaded model, i.e., memory interference and synchronization. We explain how they differ from the single-threaded model, what we added or changed during profiling and performance prediction.

7.3.1 RPPM Workflow

Figure 7.2 illustrates how RPPM is structured. The first step is to build a profile of the application's execution in a microarchitecture-independent way. Profiling is done using a Pin tool [41], a dynamic binary instrumentation tool. Some of the characteristics that we collect are the same as for the single-threaded model by Van den Steen et al. [58], i.e., statistics that relate to an individual thread's execution such as instruction mix, inter-instruction dependences and branch behavior. To model multi-threaded execution performance, we in addition need to profile synchronization behavior as well as memory system behavior.

The second step is the multi-threaded performance model itself, which operates in two phases. The first phase (Figure 7.2b) predicts the active execution time for each thread in-between synchronization events. We use the microarchitecture-independent profile to predict per-epoch active execution times for each thread. To do so, Equation 7.1 from the single-threaded model is used. Although this is the same equation, some of the numbers that serve as input to the model need to be computed differently. In particular, we need to account for the impact shared resources and cache coherence may have on per-thread performance because of a positive or negative interference.

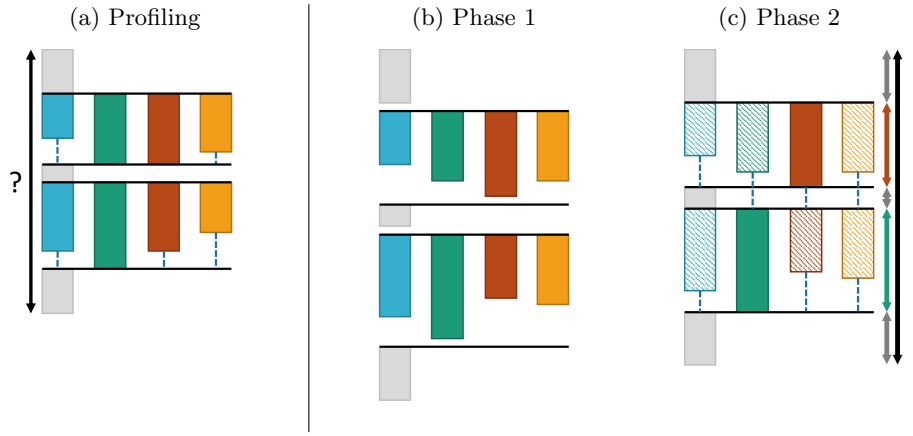


Figure 7.2: RPPM predicts multi-threaded execution time in three steps: (a) We profile an application's synchronization behavior and per-epoch statistics for each thread. We then predict an application's execution time (b) by predicting per-epoch active execution times for each active thread, and (c) by estimating the impact of synchronization on overall application performance.

The second phase, illustrated in Figure 7.2c, is a new addition compared to the single-threaded model to account for synchronization events. This phase models the synchronization behavior to predict the overhead caused by synchronization. In addition, it introduces idle time, shown as dashed lines in Figure 7.2c, to predict the overall execution time of the application.

7.3.2 Memory Interference

To model the cache behavior, we use a tool called StatStack [18]. This tool records the reuse distance between memory accesses during profiling to estimate the missrate of a fully-associative LRU cache. The single-threaded modeling tool already used StatStack to estimate the impact of memory operations on an application's execution time. RPPM is using a new version of this tool that is also capable of predicting the number of cache misses in multi-threaded applications [1].

Single-threaded StatStack

To predict the miss rate of a cache, StatStack records reuse distances between memory accesses to the same address. A reuse distance is defined as the number of accesses between two memory accesses to the same location. In the top left part of Figure 7.3, a small example where the reuse distance between the first two accesses of A equals 4; between the last two accesses of A the reuse distance equals 1. Step one is recording and merging these reuse distances into a reuse distance histogram. The result is shown in the top right part of Figure 7.3. For every reuse distance, a stack is visible. The height of the stack is

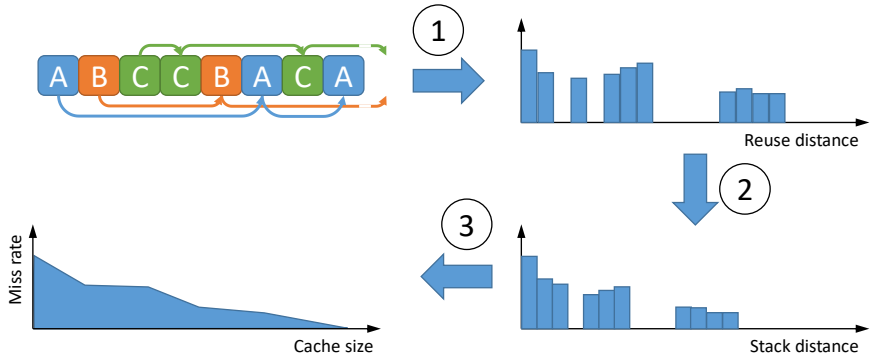


Figure 7.3: Modeling private caches is done in three steps. First profile memory accesses to create a reuse distance histogram, then derive a stack distance to predict the missrate for any cache size.

an indication of how many times this reuse distance was encountered during profiling.

The main reason reuse distance is used is because it is very easy and cheap to count the number of accesses between two memory accesses of the same data address. In a fully-associative LRU cache, an access will miss when the number of unique accesses in-between this access and the previous access to the same cache block is bigger than the size of the cache. So in order to predict whether the second access is a hit or a miss, the number of unique accesses is needed and not just the number of accesses.

Therefore, StatStack converts this reuse distance histogram into a stack distance histogram, where stack distance is defined as the number of unique accesses between two accesses to the same memory address. To convert a reuse distance into a stack distance, the number of reuse distances smaller than this reuse distance is calculated and used as an estimate for the stack distance. When this is done for every reuse distance in the histogram, the result is a stack distance histogram. Now this can easily be translated to a miss rate curve as shown in the bottom part of Figure 7.3.

Note that stack distance is in fact also microarchitecture-independent, but is a lot harder to record. Recording this would require us to keep a list of all unique accesses, which can grow rapidly leading to a large memory footprint. And we constantly need to check if an access to this address is already in the list, leading a significant slowdown during profiling.

Multi-threaded StatStack

Multiple threads of the same application can and most likely will access the same memory addresses during the execution. This can lead to both positive and negative interference depending on the organization of the cache hierarchy. An example is shown in Figure 7.4: when the two threads would run in complete isolation, the accesses to A and D would both have a reuse distance of 3 and

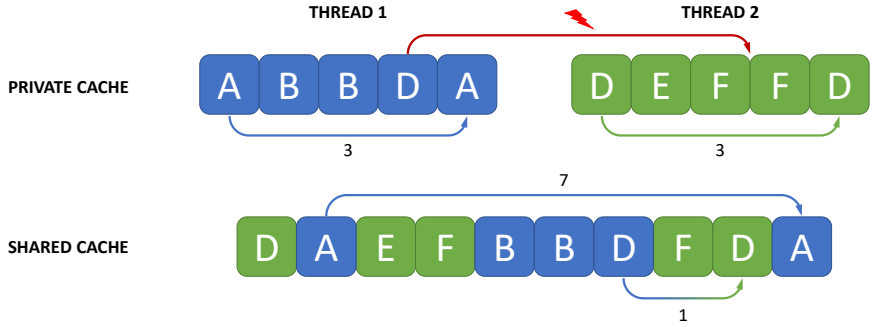


Figure 7.4: Reuse distance can change significantly when moving to a shared cache.

a stack distance of 2. But since these are two threads are from the same application, they will influence each other.

Threads can have a negative influence on each other in the private cache. As seen in Figure 7.4, the first thread is also accessing the data at memory location D. If this is a read, there is not going to be any interference with the second thread. However if thread 1 writes a new value to this memory address, this would mean that the value stored in the private cache of the second thread is no longer valid. A cache coherence protocol makes sure that the value is marked as invalid and therefore the second access to memory address D would always result in a cache miss. This is called write invalidation and it is important for the model to take this type of interference into account.

Negative interference can also happen in a shared cache. Because memory accesses of both threads interleave, there is a high possibility that the number of accesses between two memory accesses increases significantly. This happens for memory access A in Figure 7.4. Now the reuse distance increases from 3 in the private cache to 7 in the shared cache, and the stack distance increases from 2 to 4. This increase could mean that now the address is no longer in the cache.

The interference can also have a positive effect, i.e., the same access to D from the first thread that could lead to write invalidation in the private cache of the second thread, will also cause a decrease in the reuse and stack distance when sharing a cache. The reuse distance will shrink from 3 to 1; likewise, the stack distance decreases from 2 to 1. This will likely mean that this memory access will still be available in the shared cache. This can also cause some type of prefetch behavior, where one thread is always reading data for the first time, leading to cache misses, but then the data is always available in the shared cache for the other threads to access.

To model the interference in the **private cache**, StatStack records the memory accesses of all threads at the same time. By doing this, it enables StatStack to model write invalidation in the private cache. When predicting a hit or a miss for a certain memory access, it will check the ordering of accesses to the same memory location done by other threads. If there is a write in-

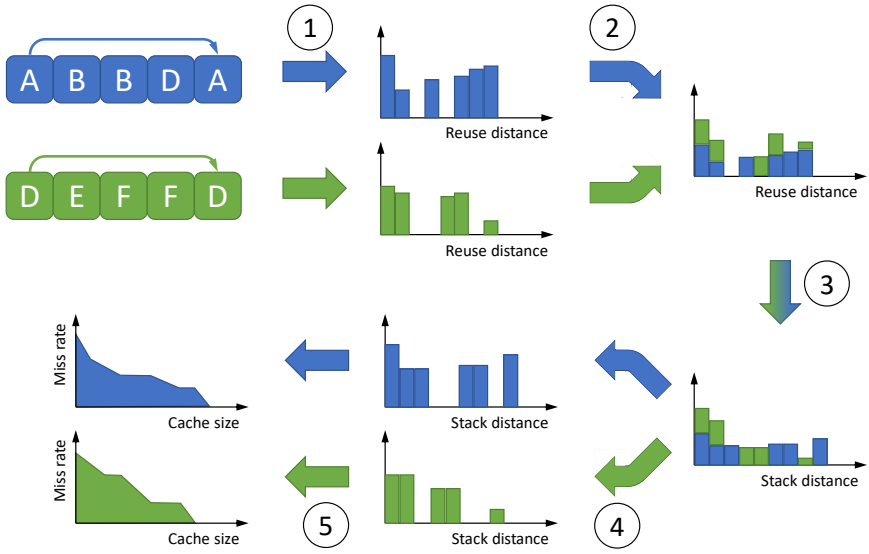


Figure 7.5: Modeling shared caches involves five steps. Two extra steps are needed to combine the individual reuse distance histograms into a shared histogram, and later split the shared stack distance histogram into histograms for every individual thread.

between, StatStack predicts a miss; if not, it will check the stack distance in the same way as for single-threaded applications (see Figure 7.4).

Modeling **shared caches** is more complicated and involves five steps as illustrated in Figure 7.5. Step one is to create reuse distance histograms for all individual threads. This is done for all threads individually since this information is needed later on in the process to create an accurate stack distance histogram per thread. But now using global reuse distances, as explained in Figure 7.4, we need to take the accesses of all other threads sharing the same cache into account. Calculating the global reuse distance is done by increasing the reuse distance with the number of accesses the other threads did in-between these two accesses. The second step is to merge the reuse distance histograms from all threads into one reuse distance histogram. Merging is done by simply stacking the histograms on top of each other. If thread 1 has 10 accesses with a reuse distance of 4 and thread 2 has 15, together they will have 25 accesses with a reuse distance of 4. This is shown in the top right corner of Figure 7.5.

Now that a merged reuse distance histogram is created, the same technique as was developed for the single-threaded version is used to derive a (combined) stack distance histogram. Although this time we are not really interested in the stack distance histogram itself, but more in the translation from reuse distance to stack distance. Now that this translation is known, we can construct a stack distance histogram for every individual thread, by translating the reuse distance into a stack distance. So if during step 3 in the process, the creation of the stack distance histogram, we learned that a reuse distance of 4 translates to a stack distance of 3, we will now construct a stack distance by adding all

accesses with a reuse distance of 4 from the first thread to stack number 3 of the stack distance histogram.

The last step (step 5 in Figure 7.5) is the creation of a miss rate curve as a function of cache size. This step is done for every individual thread and uses the same approach as for single-threaded applications. The main difference is that one thread will end up with different curves for different cache configurations: one curve for a private L1 cache, one for a shared last-level cache that is shared with all other threads, maybe a shared L2 cache that is only shared with a couple other threads, etc.

Note that StatStack assumes a particular ordering of memory accesses during profiling. We should make sure that the order in which the memory accesses appear during profiling is representative of the order in which the memory accesses would appear in the modeled hardware configuration.

7.3.3 Synchronization

Synchronization is used by multi-threaded applications for different goals. Threads of an application should be synchronized to avoid conflicts in critical resources, or to make sure threads make equal progress, or to wait for input from other threads, etc.

We track all synchronization events (barriers, critical sections, condition variables, etc.) by tracking specific library function calls. The current version of RPPM provides support for both the pthread and OpenMP parallel execution models, but even user-created solutions that do not use an existing library could be implemented with minimal effort. This can be done by just adding the code into a function and communicate to RPPM to track all calls to this function and model it accordingly.

When using the pthread library, the programmer typically uses the available function calls to mark synchronization events. Defining a critical section for example is done by calling `pthread_mutex_lock` and `pthread_mutex_unlock` at the start and end of a critical section, respectively. For OpenMP, the programmer marks a `for` loop with a `#pragma` telling the compiler to insert the necessary function calls for the runtime environment to execute the loop using parallel threads. For example, the function call to `gomp_team_barrier_wait` marks a barrier. We capture these function calls in the profiler and log the location of the calls in the application's synchronization profile.

Complex parallel applications use multiple barriers and/or multiple critical sections. To be able to distinguish different synchronization events, we also track function arguments. For example, the function `gomp_team_barrier_wait` passes the barrier (`gomp_barrier_t`) as a pointer, and by tracking these function arguments we keep track of which specific barrier a thread is waiting for.

Condition Variables

Condition variables are a widely used synchronization primitive but require special support beyond instrumenting function calls to the synchronization library. In the Parsec benchmarks, condition variables are used to implement barriers and producer-consumer relations.

To fully understand the examples in Algorithms 2 and 3, a short explanation for the used function calls is useful:

`pthread_mutex_lock(&mutex):` Since different threads are accessing the same condition variable, this needs to happen in a critical section. A critical section is a region of code that only one thread can access at the same time. To make sure that only one thread has access to this region at the same time, they need to lock the section by calling the `pthread_mutex_lock` function. Only the first thread gets the lock, the other threads need to wait until this thread finishes its execution of the critical section. The `&mutex` argument is used to distinguish between different critical sections.

`pthread_mutex_unlock(&mutex):` This function call releases the obtained lock and is used by the thread to indicate that it finished the work in the critical section and leaves the critical section to be accessed by another thread.

`pthread_cond_wait(&cond, &mutex):` When the condition is not met, a thread calls the `pthread_cond_wait` function to tell the pthread library that it cannot make any progress. It releases the critical section (indicated by the `&mutex` argument) and stalls the execution. The thread stalls until another thread indicates that the condition has changed and that now the thread might be able to make progress again. When this happens, the thread tries to lock the critical section again and continues the execution in the critical section. The `&cond` argument is used to distinguish between different condition variables.

`pthread_cond_broadcast(&cond):` A thread calls the `pthread_cond_broadcast` function to wake up all threads that are waiting for this condition to change. Calling this function never results in a stall, so releasing the critical section is not needed.

These four function calls are used by the programmer to implement different synchronization techniques. A barrier and producer-consumer synchronization are the most frequently used in the Parsec benchmarks. These two are now discussed in detail, but they can also implement semaphores and other variations.

Barriers. Condition variables are frequently used to implement barriers, for which the variable is used to count the number of threads that have

Algorithm 2 Barrier using condition variables.

```
1: pthread_mutex_lock(&mutex);
2: n++;
   marker_cond_wait(&cond, &mutex);
3: if n < threads then
4:     pthread_cond_wait(&cond, &mutex);
5: n = 0;
6: pthread_cond_broadcast(&cond);
7: pthread_mutex_unlock(&mutex);
```

reached the barrier. One way of implementing this, is shown in Algorithm 2. All actions are done within a critical section. After successfully grabbing the lock and entering the critical section, the thread will increment the condition variable (n). Next, it checks whether the variable equals the number of worker threads. If not, the thread calls the `pthread_cond_wait` function, pauses its execution and releases the lock. If the condition is met, the thread calls the `pthread_cond_broadcast` function to tell all waiting threads that the condition is satisfied and all threads can continue their execution.

As mentioned before, we profile all pthread library calls to characterize an application's synchronization behavior. Unfortunately, this is not sufficient for condition variables because the `pthread_cond_wait` function is not always called. In particular, the last thread arriving at the barrier does not actually call this function, see Algorithm 2. The problem here is that which thread arrives at the barrier the latest, or in other words, which thread does not call the `pthread_cond_wait` function, depends on the micro-architecture on which the application is executed, and may be different between the profiling run and the run for which the model predicts performance. To be able to adequately model condition variables, we need to know when there is a 'possibility' for a thread to wait — not only when the thread effectively waits during the profiling run. We therefore introduce a marker between lines 2 and 3 in Algorithm 2 to notify the profiler that all threads can potentially call the `pthread_cond_wait` function. This allows RPPM to capture the condition variable for all threads.

Producer-Consumer. A second synchronization technique that is implemented using condition variables is a producer-consumer relationship. A producer-consumer relationship is used when on thread (the consumer) needs to wait for another thread (the producer) to finish its execution and produce a result that the consumer needs to continue. An example of a producer-consumer algorithm is shown in Algorithm 3.

The producer increments the condition variable i to indicate it produced an item or job ready for a consumer to process. To signal any waiting consumers that it produced an item, the thread calls the `pthread_cond_broadcast` function. The code would work without the `if`-statement (on line 2) as

Algorithm 3 Producer-Consumer using condition variables.

Producer:

```
1: pthread_mutex_lock(&mutex);
   marker_cond_broadcast(&cond);
2: if i == 0 then
3:     pthread_cond_broadcast(&cond);
4: i++;
5: pthread_mutex_unlock(&mutex);
```

Consumer:

```
1: pthread_mutex_lock(&mutex);
   marker_cond_wait(&cond, &mutex);
2: while i == 0 do
3:     pthread_cond_wait(&cond, &mutex);
4: i−−;
5: pthread_mutex_unlock(&mutex);
```

well, but some of the benchmarks first check if the number of items is zero. If it is, there is a possibility that there are consumers waiting. If i is not zero, then it is impossible for a consumer to be waiting, thus calling the broadcast function would be unnecessary overhead.

Although the code would work just fine without the `if`-statement, since there is an `if`-statement, we need to add a marker between lines 1 and 2 (`marker_cond_broadcast`) to indicate that there is a possible call to the `pthread_cond_broadcast` function. The number of times the actual broadcast function is called depends on the behavior of the application during profiling, therefore, the marker can be used instead.

The consumer first checks if there are items left. If there are no items to process, it calls the `pthread_cond_wait` function and waits for a producer to call the broadcast function. If there are items left, the consumer does not need to wait and just takes an item to process and indicates this by decrementing the number of available items (i). Just like in the case of a barrier, whether or not the wait and broadcast functions are called depends on the architecture during the profiling run. To profile this possible wait location, we add a marker between lines 1 and 2 in the source code.

The problems with conditionally calling synchronization functions is solved by adding markers in the source code and by catching them during profiling. While this involves manual changes to the source code, it is not that cumbersome in practice: searching the respective condition variable function calls and adding markers is fairly straightforward. For our set of benchmarks from Parsec (we used the OpenMP version of Rodinia), we have five, out of 10, benchmarks that use condition variables. For four benchmarks (bodytrack, raytrace, streamcluster and vips), we had to add one marker for the `pthread_cond_wait`

function. For facesim, we added a marker for the `pthread_cond_wait` function and a marker for the `pthread_cond_broadcast` function. In total, we thus had to add six markers.

Estimating synchronization overhead

The overall execution time of a multi-threaded application is predicted by combining the predicted per-epoch active execution times for each of the threads with the predicted synchronization overhead.

Algorithm 4 Estimating synchronization overhead

```
1: while not finished do
2:     for Thread T in sorted(Threads, shortestTimeFirst()) do
3:         if not isBlocked(T) then
4:             Proceed T to next synchronization event
5:         else
6:             Add idle time
```

Estimating synchronization overhead is done using Algorithm 4. We identify the thread with the shortest total execution time (active and idle time) thus far that is not blocked by the next synchronization event and symbolically proceed it to this next event. We emulate the execution behavior at each synchronization event and we repeat this process until all threads reach the end of execution and the application finishes. At the end of the symbolic execution, the critical path through the execution determines the application's execution time.

During the symbolic execution while emulating a synchronization event, we calculate the number of cycles a thread spends waiting for other threads, not making forward progress. This is illustrated in Figure 7.2(c) for barrier synchronization. Active execution time is depicted by a box; waiting time is depicted by a dashed line; overall execution time is determined by the slowest thread in-between synchronization events. In particular, the execution time of the first inter-barrier epoch is determined by the third thread; the execution time of the second inter-barrier epoch is determined by the second thread; overall execution time is predicted by summing up the predicted inter-barrier execution time and the main thread's execution times when it is running alone.

We account for the different synchronization events as follows:

- Thread creation: The main thread is created at application start-up time; all other threads are therefore initially marked as 'blocked'. When the main thread creates a new thread, this thread is 'unblocked' and its start time is set accordingly.
- Critical sections: A critical section is a code segment that has to be executed atomically, by one thread at a time. We mark accessing and leaving a critical section as a synchronization event. Before a thread is allowed to enter a

critical section, the symbolic execution verifies that no other thread is currently executing that same critical section. If so, the thread blocks waiting for the critical section to be released. Once released, the thread is allowed to proceed and enter the critical section. The waiting time and the actual execution time of the critical section determines overall execution time.

- Barriers: A barrier is a location in the code where all threads need to wait for each other to finish the execution of their respective code segment. A thread can only continue when all threads have reached the barrier. When a thread arrives at a barrier it checks whether all other threads already reached the barrier. When this is not the case, the thread blocks itself and waits. The last thread arriving at the barrier releases the barrier and determines the execution time of the inter-barrier epoch.

- Condition variables: As mentioned in Section 7.3.3, we add markers to catch condition variables during profiling. We use these markers to verify the intended behavior of the condition variable. If the condition variable is used to implement barrier synchronization — easily recognized if all but one of the threads need to wait at the condition variable and any of the threads can release the barrier — we model the condition variable as a barrier, as just described. A producer-consumer relationship is recognized if a thread or set of threads wait at the synchronization event, i.e., the consumer thread(s), and another thread or set of threads calls the broadcast function to release the waiting thread(s), i.e., the producer thread(s). Waiting at the synchronization event may be conditional, i.e., threads only have to wait if there are no items to process. The broadcast operation may be conditional as well, i.e., the producer may only broadcast new items if at least one consumer is waiting for a new item. The producer-consumer relationship is modeled by keeping track of the number of broadcast markers, i.e., the number of created items. If the number of created items equals zero at the time a consumer reaches the synchronization event, the consumer threads is stalled. As soon as the number of items is larger than one, the consumer thread resumes its execution.

- Thread joining: A join is called when waiting for a thread to terminate. The behavior is similar to a barrier, i.e., the execution time of the longest running thread determines when the join happens. The difference in execution time is added as idle time to the shortest running thread.

We note that this is not a complete list of all possible synchronization events, but a list of all events encountered in our benchmark suite. Nevertheless, we are convinced that this approach will be suitable for unlisted events like semaphores or even indirect synchronization.

7.4 Experimental Methodology

We now describe the methodology used to evaluate the RPPM model.

Benchmark	Input	Instructions [10^6]	Barriers
Backprop	4194304	503	8
BFS	graph8M	231	52
CFD	fvcorr.domn.010K	55 244	32 000
Heartwall	test.avi 10	72 055	20
Hotspot	16384 5	39 220	10
Kmeans	kdd_cup	29 983	74
LavaMD	10	224 944	2
Leukocyte	testfile.avi 5	96 455	14
LUD	2048.dat	34 165	508
Myocyte	100 100 1	9 627	2
NN	4096k	6 637	8 194
NW	16k x 16k	6 587	4 096
Particlefilter	128 x 128 x 10	3 501	148
Pathfinder	1M x 1k	22 979	1 998
SRAD	2048	2 879	20
Streamcluster	256k	199 620	3 262

Table 7.2: Overview of all Rodinia benchmarks used.

7.4.1 Benchmarks

Rodinia3

We consider all the benchmarks from the Rodinia benchmark suite v3.1 [11]. The benchmarks are listed in Table 7.2 along with the used input and the number of dynamically executed instructions. As can be seen from the table, the inputs where chosen such that the number of instructions executed within the region of interest falls between 250 million and 250 billion. The region of interest or ROI is defined as the region which starts after the initialization and ends just before the finalization by the main thread; during the ROI, multiple threads co-execute.

We use the OpenMP implementation of the Rodinia benchmarks. When using OpenMP, the programmer marks a `for` loop with a `#pragma` telling the compiler that this loop should be executed in parallel. The compiler divides the work among all threads (including the main thread) and inserts barriers during the execution to make sure threads make equal progress. The number of barriers executed varies between 2 and 32,000. The per-benchmark statistics are shown in the last column of Table 7.2.

All our experiments are done using 4 threads. This can be set by using an environment variable called `OMP_NUM_THREADS`. When setting this value to 4, OpenMP creates a pool of worker threads containing 3 threads. During execution, the main thread is also executing its share of the work. Hence, during the parallel region, 4 threads are active.

Benchmark	Instructions [10^6]	Synchronization events		
		Critical Sections	Barriers	Cond. var.
Blackscholes	1 032	-	-	-
Bodytrack	3 131	6700	98	25
Canneal	827	4	64	-
Facesim	21 800	10472	-	1232
Fluidanimate	3 023	2140206	50	-
Freqmine	8 774	-	-	-
Raytrace	3 107	47	-	15
Streamcluster	4 339	68	13003	34
Swaptions	7 298	-	-	-
Vips	11 806	8973	-	1433

Table 7.3: Overview of all Parsec benchmarks used.

Parsec3

We are using all the applications and kernels from the Parsec version 3.0 benchmark suite [4]. They are listed in Table 7.3 along with the number of dynamically executed instructions. For this benchmark suite, inputs are provided and we choose the `simmedium` inputs. This results in a dynamic instruction count ranging from just under 1 billion to 20 billion instructions.

For the Parsec benchmarks we use the pthread implementation in order to be able to evaluate both the pthread and OpenMP parallel execution models. Apart from barriers, these benchmarks also use critical sections and condition variables as synchronization primitives. The number of events encountered during the execution is shown in the rightmost column of Table 7.3. Three benchmarks (blackscholes, freqmine and swaptions) do not have any synchronization during the ROI, so the only location where synchronization overhead can be introduced is at the very end of the ROI where all threads wait for the last one to finish.

During our evaluation we are setting the number of cores to 4. All of the Rodinia benchmarks will create a pool of 3 worker threads and together with the main thread this makes 4 threads in total. The Parsec benchmarks spawn more threads, however, parallelism is limited to 4, i.e., this means at any given time, only 4 threads will simultaneously run.

Since our model assumes that threads are running in isolation, and thus not taking into account any form of time sharing, we adjust this number so that there are only 4 threads with an impact on the execution time[1]. Therefore the number of threads was decreased to 3 for bodytrack, streamcluster and vips. Dedup and ferret were excluded since there is no setting that only spawns 4 threads with significant impact on execution time. Furthermore, x264 was excluded since this benchmark is not running on our simulation environment.

[1] An impact means that this threads is active for more than $> 1.0\%$ of the complete execution time.

	Smallest	Small	**Base**	Big	Biggest
frequency [GHz]	5.00	3.33	2.50	2.00	1.66
dispatch width	2	3	4	5	6
ROB size	32	72	128	200	288
issue queue size	16	36	64	100	144
branch predictor	4 KB, tournament				
L1-I cache	32 KB, 4-way, private				
L1-D cache	32 KB, 4-way, private				
L2 cache	256 KB, 8-way, private				
LLC	8 MB, 16-way, shared				

Table 7.4: Simulated architecture configurations.

7.4.2 Simulator

We evaluate RPPM's accuracy as follows. We first simulate the benchmarks using the Sniper multicore simulator [8], which is a state-of-the-art, parallel and hardware-validated multicore simulator. We simulate the Base multicore configuration as specified in Table 7.4, unless mentioned otherwise. These simulated execution times serve as the golden reference.

7.4.3 Profiling

We also profile the benchmarks and subsequently predict execution time for our benchmarks using RPPM for the exact same multicore architecture that we simulated using Sniper. We then compute the error between the simulated and predicted execution times. Profiling is done using four of threads on an Intel Xeon Gold 6140.

7.5 Evaluation

We evaluate RPPM's accuracy against cycle-level simulation and compare against two naive extensions of the previously proposed single-threaded performance model, MAIN and CRIT, as previously described in Section 7.2.3. The results are shown in Figure 7.6, with the Parsec benchmarks in the top row and the Rodinia benchmarks in the bottom row; averages are reported on the far right.

MAIN predicts the execution time of the main thread to predict overall application performance. This leads to an average absolute prediction error of 45% with several outliers up to 100%. The outliers are more common for the Parsec benchmarks because the main thread is just doing some bookkeeping and not performing any actual work. This leads to a significant underestimation of the application's overall execution time.

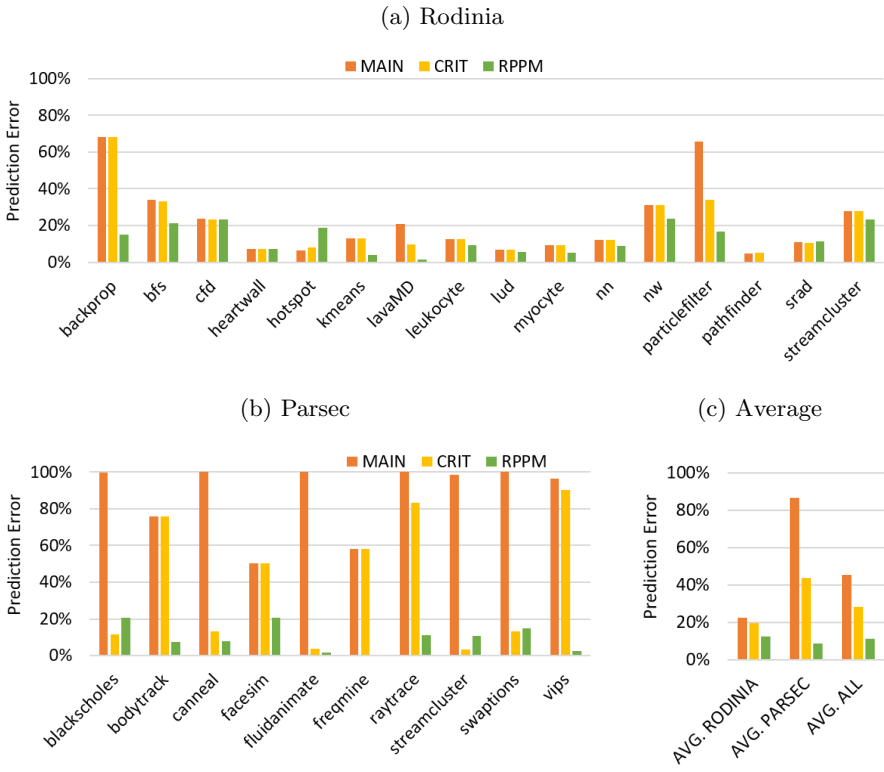


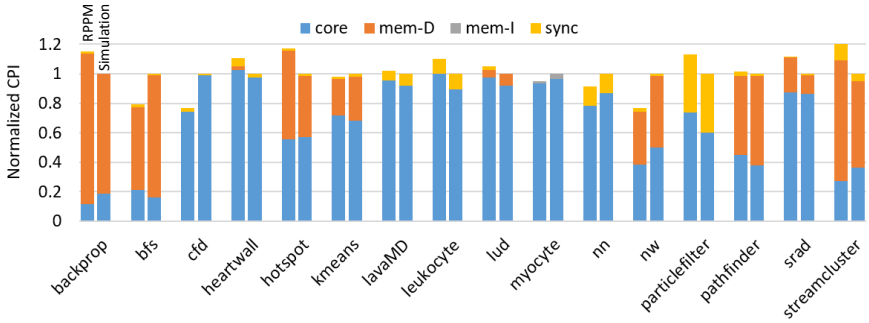
Figure 7.6: Prediction error for MAIN, CRIT and RPPM.

CRIT predicts the execution time for all threads and then takes the execution time of the slowest thread (critical thread) as a prediction for the application's execution time. CRIT reduces the prediction error to 28% on average. CRIT is more accurate than MAIN, particularly for the Parsec benchmarks, because CRIT models the worker threads as opposed to just modeling the main thread as done by MAIN.

RPPM clearly outperforms MAIN and CRIT with an average absolute error of 11.2% and a maximum error of 23%. RPPM accurately predicts which thread is the most critical thread in-between synchronization events which leads to an overall more accurate prediction than MAIN and CRIT.

To help understand where the remaining error is coming from, Figure 7.7 illustrates the average per-thread normalized CPI stacks. We measure average per-thread CPI by computing the respective CPI stacks for each thread separately and then compute the average. RPPM's modeling error is due to inaccurate predictions for the base component (e.g., cfd), the mem-D component (e.g., backprop) or both (e.g., nw). These inaccuracies originate from the single-threaded prediction model and/or the extended memory hierarchy model, which indirectly leads to incorrect predictions for the synchronization component.

(a) Rodinia



(b) Parsec

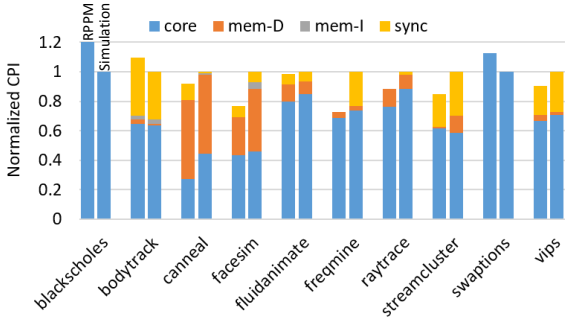


Figure 7.7: CPI stacks for RPPM (left) and simulation (right) normalized to simulation.

Chapter 8

Behavior of Multi-Threaded Applications

Having evaluated the accuracy of RPPM for predicting multi-threaded application performance on multicore hardware, we now consider a couple case studies to illustrate the usefulness of RPPM for driving design space exploration studies and workload behavior analyses.

8.1 Design Space Exploration

We now consider the following case study to illustrate RPPM's usefulness for exploring complex design trade-offs. We profile each of the benchmarks once and predict performance for five different configurations as listed in Table 7.4. We change processor width from 2 to 6 (and scale ROB and issue queue resources accordingly) and change clock frequency from 5 to 1.66 GHz across these design points, while keeping the maximum number of operations that can be executed per second constant.

We use RPPM to identify the design points that are within a bound of x% of the predicted optimum, see Table 8.1. If the bound is set to 0%, only the best design point is identified by RPPM. If the bound is larger than 0%, all design points within the bound are identified by RPPM and simulation will select the best one. The average deficiency (performance difference) versus the real optimum is 1.95% (see bottom row) and up to 19.1% for streamcluster. Setting a higher bound of 5% increases the number of predicted optimum design points (up to 2 for some benchmarks, see rightmost column) but brings down the deficiency of the identified design points to the true optimum to at most 1.97% for pathfinder.

Bound	**0%**		**< 1%**		**< 3%**		**< 5%**	
backprop	0.00%	1	0.00%	1	0.00%	1	0.00%	1
bfs	0.00%	1	0.00%	1	0.00%	1	0.00%	2
cfd	0.00%	1	0.00%	1	0.00%	1	0.00%	1
heartwall	0.00%	1	0.00%	1	0.00%	1	0.00%	1
hotspot	0.00%	1	0.00%	1	0.00%	1	0.00%	1
kmeans	0.00%	1	0.00%	1	0.00%	1	0.00%	2
lavaMD	0.00%	1	0.00%	1	0.00%	1	0.00%	1
leukocyte	0.00%	1	0.00%	1	0.00%	1	0.00%	1
lud	0.00%	1	0.00%	1	0.00%	1	0.00%	1
myocyte	0.00%	1	0.00%	1	0.00%	1	0.00%	1
nn	0.00%	1	0.00%	1	0.00%	1	0.00%	1
nw	10.15%	1	10.15%	1	10.15%	1	0.00%	2
particlefilter	0.00%	1	0.00%	1	0.00%	1	0.00%	1
pathfinder	1.97%	1	1.97%	1	1.97%	1	1.97%	2
srad	0.00%	1	0.00%	1	0.00%	1	0.00%	1
streamcluster	19.11%	1	0.00%	2	0.00%	2	0.00%	2
average	1.95%		0.76%		0.76%		0.12%	

Table 8.1: Case study: Predicting the optimum design point.

8.2 Bottle Graphs

Bottle graphs are a useful tool for analyzing multi-threaded application performance [16]. The bottle is formed by a stack of boxes, where every box represents one thread of the application. The height of the box shows the criticality or the impact of that thread on the overall execution time of the application.

The criticality of a thread is defined as the execution time divided by the number of running threads. To calculate criticality, the execution of the application is divided into intervals (i) where the number of running threads (r_i) is constant, the resulting criticality is the average criticality for every interval. Equation 8.1 shows how to calculate criticality for thread j, t_i represents the length of every interval.

$$C_j = \sum_{\forall i} \frac{t_i}{r_i} \tag{8.1}$$

The width of the box in a bottle graphs also shows the level of parallelism of that thread. The parallelism of a thread is defined as the weighted harmonic mean of the number of running threads for every interval where this thread was active. To form the bottle graph all boxes are stacked upon each other, starting from the most parallel thread (widest box) on the bottom to the one with the smallest parallelism on top. In some cases, boxes are so thin they are not visible in the bottle, in that case the thread has a negligible impact on performance.

In other words, the bottle graph of a perfect parallel application would be a stack of boxes with equal height and a width equal to the number of threads. The bottle of a sequential application would be a single box with a height equal to the total execution time and a width of 1.

See Figure 8.1 for the bottle graphs of all Parsec benchmarks: the left side of the bottle is based on data provided by the RPPM model; the right side of the bottle represents data obtained through simulation. The key conclusion is that our model is able to give an accurate representation of the thread behavior of the applications.

During execution of the benchmarks, we set the number of threads to 4. Although 7 benchmarks actually use more than 4 threads, there are no benchmarks with a parallelism larger than 4, meaning that there are no benchmarks that use more than 4 threads simultaneously. As discussed in Section 7.4, we tuned the setting for the Parsec benchmarks such that there are no benchmarks that have more than 4 threads with a considerable impact on execution time. This can be seen in the bottle graphs because there are no benchmarks with more than 4 boxes with a significant ($> 1.0\%$) height.

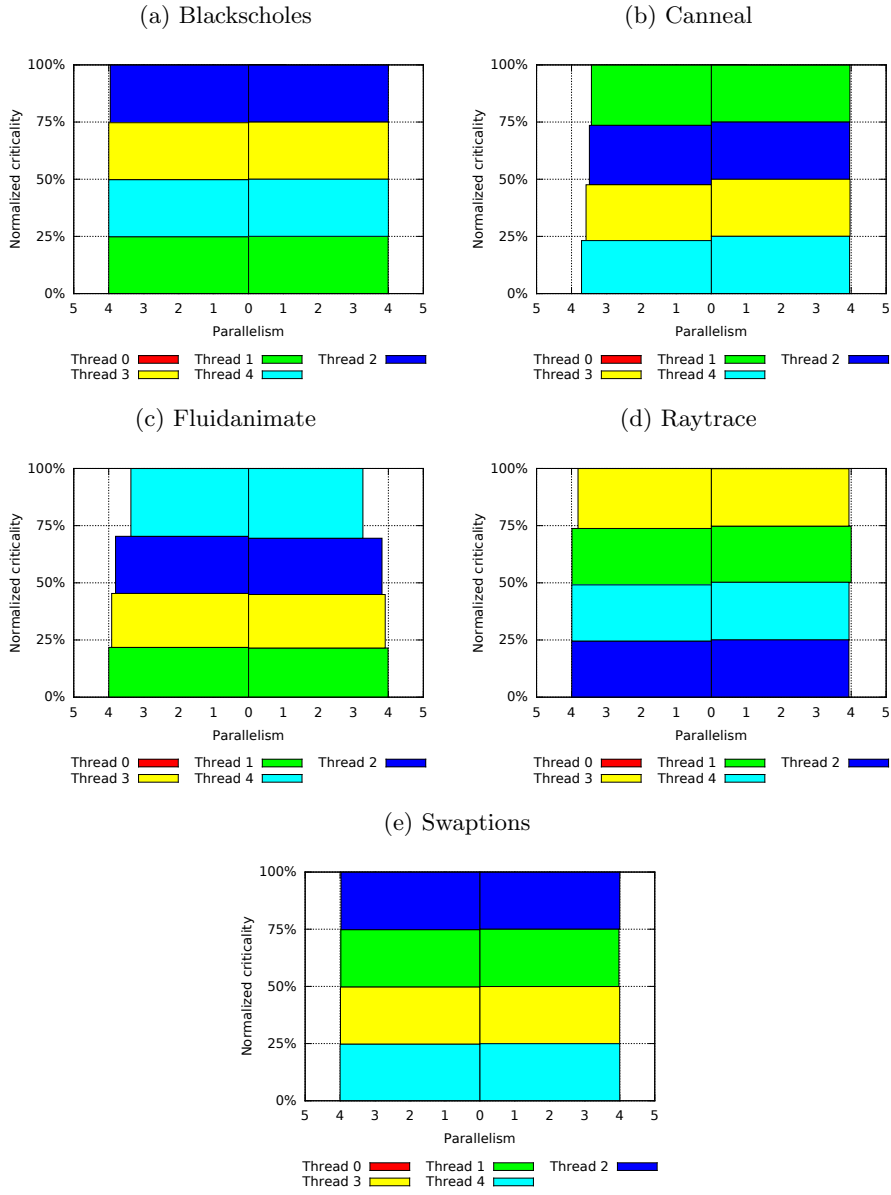


Figure 8.1: Bottle graphs for all Parsec benchmarks (part 1): the left half of the bottle graph is obtained using RPPM whereas the right half is obtained through simulation.

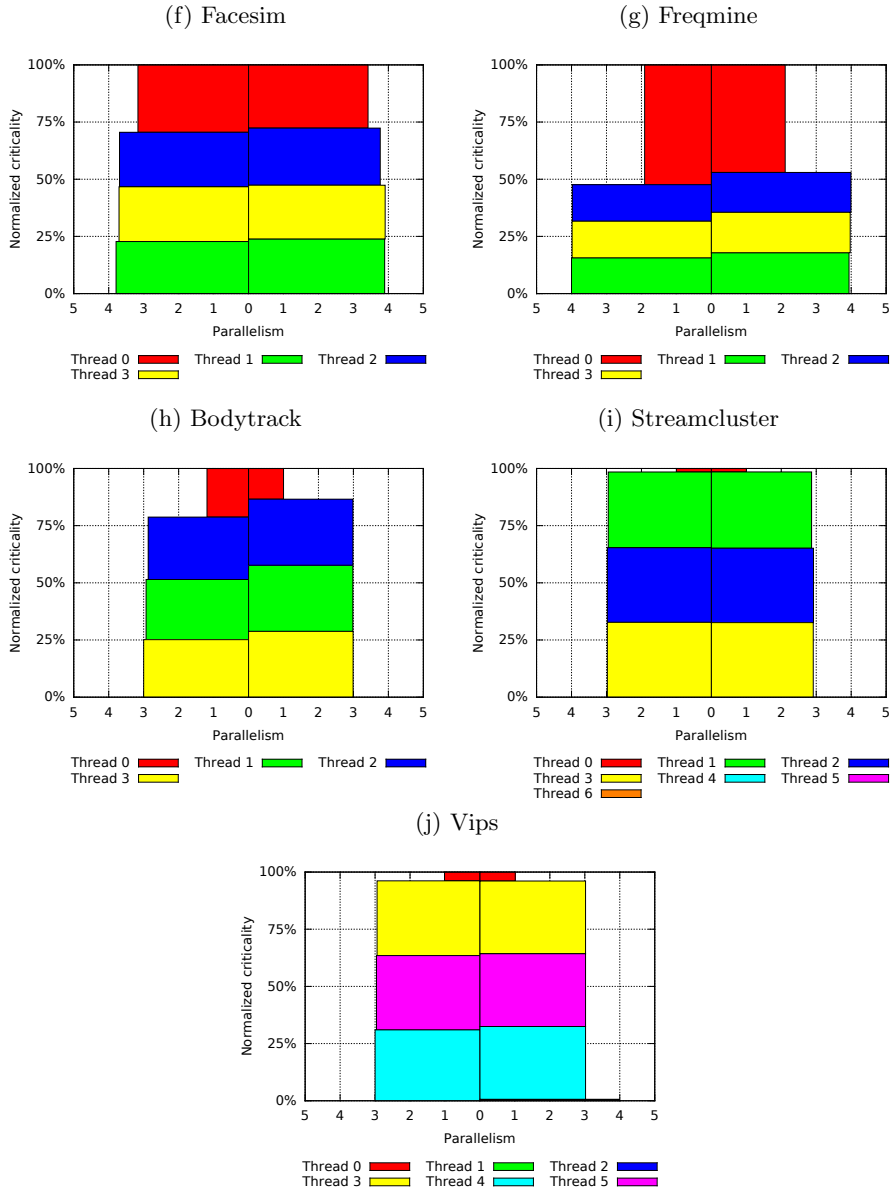


Figure 8.1: Bottle graphs for the Parsec benchmarks (part 2): the left half of the bottle graph is obtained using RPPM whereas the right half is obtained through simulation.

Based on these bottle graphs, we can divide the benchmarks into three groups:

Figures 8.1a to 8.1e: These first five benchmarks are benchmarks for which the main thread is not doing any actual work. The main thread will create 4 worker threads, which divide the work in a very balanced way. This can be seen because the shape of all boxes is very similar and very close to the maximum parallelism of 4.

Figures 8.1f and 8.1g: The following two benchmarks are using 4 threads: the main thread creates 3 worker threads and participates in the actual work. Facesim is able to achieve a near balanced bottle, but there is a clear sequential phase for the main thread. This can be seen because the maximum parallelism of the main thread (thread 0) is considerably smaller than 4.

Figures 8.1h to 8.1j: The last three benchmarks are the ones where we decreased the maximum parallelism to 3 because otherwise there would be more than 4 threads with a significant impact on overall execution time. The main thread of these benchmarks is not doing any of the work, but still has an impact between 11% for bodytrack and 1.5% in the case of streamcluster.

8.3 Time Sharing

As mentioned in Section 7.4, we left dedup and ferret out of the evaluation since there is no setting that results in only 4 threads with a significant impact on performance. This is important since the model does not take any form of time sharing into account. When RPPM predicts the execution time, it assumes that threads are not time-sharing the core resources. When RPPM estimates the execution time for these benchmarks and we compare it against a simulation of a quad-core architecture the errors are 18% and 57% for dedup and ferret, respectively. This is shown in Table 8.2. Both of our predictions are a severe underestimation of the result obtained through detailed simulation.

This is because we are not modeling the overhead of multiple threads sharing a core. If we rerun the simulation and scale the simulated architecture by adding as many cores as the application is creating threads, we make sure the threads run in isolation again. These new numbers are shown in the last column of Table 8.2. As can be seen, the prediction errors are now only 0.4% and 1.6% for dedup and ferret, respectively. This shows that when the impact of time sharing is removed, the model performs very well.

	RPPM	**Simulation**			
		4 cores		isolation	
dedup	231 ms	283 ms	18.4%	230 ms	0.4%
ferret	242 ms	568 ms	57.4%	246 ms	1.6%

Table 8.2: Impact of time sharing on dedup and ferret.

So when running the threads in isolation, the total execution time is predicted with high accuracy, but that is only one part of what RPPM can do. Using bottle graphs, RPPM can also be used to visualize the synchronization behavior of the benchmarks, as we will discuss now for both dedup and ferret

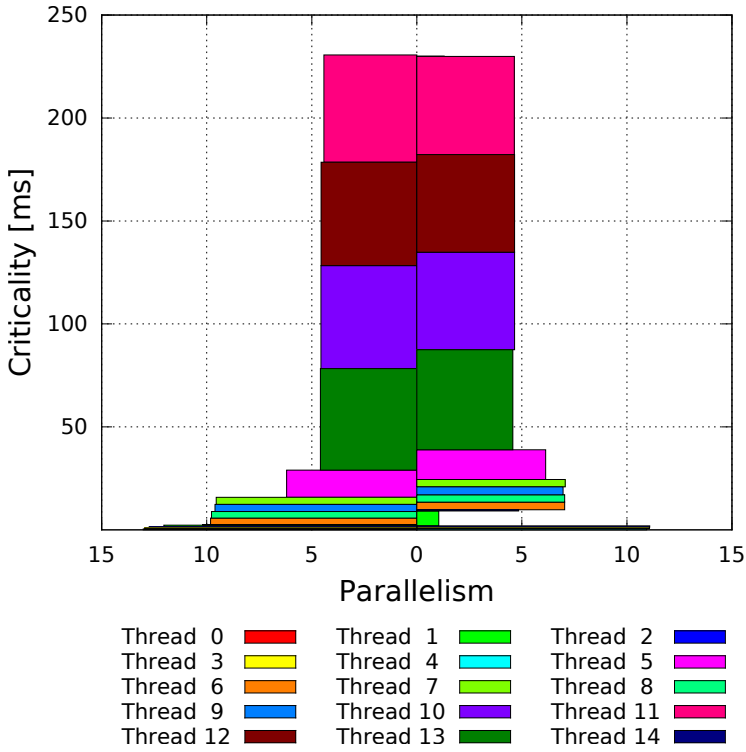


Figure 8.2: Bottle graph for the dedup benchmark. The left half of the bottle graph is obtained using RPPM whereas the right half is obtained through simulation.

Dedup: Figure 8.2 shows the bottle graph for the dedup benchmark: on the left, the result derived from RPPM is shown; on the right, the bottle graph derived from simulation is shown. Most of the work is done by threads 10, 11, 12 and 13. The boxes (threads) are ordered following the criticality as predicted through RPPM. The behavior predicted by RPPM is almost identical to simulation.

Note though there is some non-deterministic behavior. During some runs, the impact of thread 5 was negligible and the work of thread 5 was done by threads 3 or 4. Therefore, we had to rerun dedup a couple times to have a comparable execution to create a similar bottle graph with similar work done by the same threads.

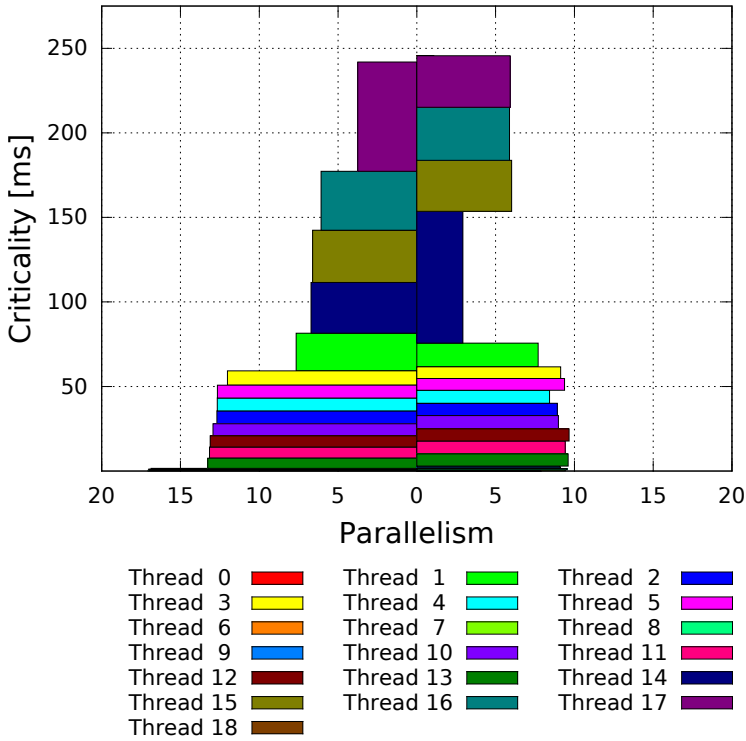


Figure 8.3: Bottle graph for the ferret benchmark. The left half of the bottle graph is obtained using RPPM whereas the right half is obtained through simulation.

Ferret: Figure 8.3 shows similar RPPM and simulation bottle graphs for the ferret benchmark. A lot of threads have a significant impact on the execution time; the top 4 threads only account for 66% of the complete execution time. Just as for dedup, there is also some non-determinism seen in the bottle graph between threads 14 and 17, where the parallelism is limited for thread 17 during the profiling run and limited for thread 14 during simulation.

Another fact that is clearly highlighted by the bottle graph is that all threads, but one, have a parallelism higher than 4. This may imply that ferret can benefit a lot from increasing the available cores. This is affirmed by the drop in execution time seen in Table 8.2 when all threads run in isolation.

8.4 Parallel Execution Models

The streamcluster benchmark is available in both the Parsec and the Rodinia benchmark suites, so this gives us the opportunity to perform a case study about the difference between the OpenMP implementation from Rodinia and the pthread implementation from Parsec. Both implementations are shown as a split bottle graph in Figure 8.4. OpenMP utilizes the main thread both for the initialization like creating a pool of worker threads, and for doing actual work. This leads to a bottle graph with the main thread having a parallelism of 2, and the worker threads achieving a near perfect (4) parallelism, meaning that when these worker threads are running, there are four threads active at the same time. This is in contrast with the pthread version, where the worker threads are only achieving a parallelism of 3 and the impact of the main thread on the execution time is only 1.5%.

Although the OpenMP version is using the available threads in a more optimal way, it is only able to decrease the execution time of the application by 3%. It looks like OpenMP is introducing significant overhead that almost completely removes the potential improvement of using all available threads in a more optimal way. It is clear from Figure 8.4 that OpenMP introduces a lot of sequential overhead in the main thread. This can be seen by the decreased width of the red box. This box can be split into a pure sequential and perfect parallel region, represented by the dashed line.

The height of the parallel region is almost identical compared to the other threads, so the parallel work is equally divided among all threads. But when looking at the sequential region, it is clear that this region takes a lot longer (taller box) to execute than the sequential main thread of the pthread implementation. This is due to the synchronization overhead that the main thread is doing. All OpenMP applications from the Rodinia suite behave just like the example from Figure 7.2. After every parallel region there is a sequential region where only the main thread is active. Although this region is never very long, but since this is a sequential region, the criticality is very high.

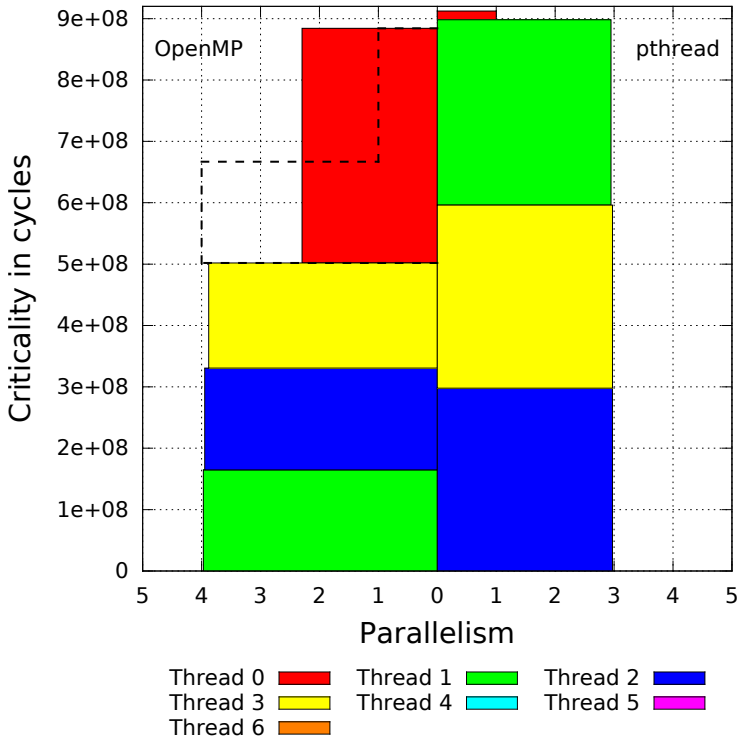


Figure 8.4: Bottle graph for the OpenMP implementation on the left versus the pthread implementation on the right for the streamcluster benchmark.

Chapter 9

Conclusions and Future Work

This chapter summarizes the main conclusions of this thesis and gives some reflections on potential future work.

9.1 Linear Branch Entropy

In the first part of this thesis we proposed linear branch entropy as a means to characterize the branch behavior of an application in a micro-architectural independent way. Linear branch entropy correlates well with misprediction rates, which enables building a simple linear model that estimates branch misprediction rates for different branch predictors using a single branch entropy profile of an application. The model is more accurate than the previously proposed branch classification methods, and allows for exploring a large design space using a single profiling run, as opposed to performing multiple branch predictor simulations.

We illustrated how branch entropy can be used to classify and select benchmarks. Furthermore, the branch predictor model allows for comparing branch predictors. In particular, we analyzed the top-four predictors of the latest branch predictor competition. By comparing the models from these branch predictors, we were able to show that the third runner-up is achieving higher accuracy when predicting the outcome for hard-to-predict branches.

We leverage linear branch entropy to create a set of benchmarks with well-balanced branch behavior. Our proposed technique was later used to select the benchmarks for the Championship Branch Competition (CBP) in 2016.

Linear branch entropy is a suitable and accurate metric to be used in a micro-architectural independent profiling tool to predict the execution time of single- or multi-threaded applications. To model these applications, linear

branch entropy can be used to estimate the number of mispredicted branches, which in turn can be used to predict the impact of the branch behavior on overall execution time.

If-conversion needs to be done meticulously, because some branches show a performance benefit when if-converted, while if-converting other branches leads to a performance degradation. We postulate that hard-to-predict branches benefit more from if-conversion by avoiding the branch misprediction penalty. We proposed a new criterion based on branch entropy to guide if-conversion in a branch predictor agnostic way, and we show that it leads to significant performance benefits of up to 2.4% compared to standard if-conversion.

9.2 Multi-Threaded Performance Modeling

In the second part of this thesis, we proposed RPPM, which takes micro-architecture-independent characteristics as input to predict performance of multi-threaded applications on a previously unseen multicore platform. These characteristics are first measured during a profiling run. This profile measures instruction count, instruction mix, branch entropy, reuse distances between memory accesses, etc. These characteristics are then used as input to a mathematical model to predict the execution time of the multi-threaded application on a target multicore processor.

RPPM extends prior work that was aimed at modeling single-threaded application performance. This prior work is used in the first phase of the prediction model. During this phase, the methodology of the single-threaded model is used to model per-epoch active execution times per thread. To accurately model interference in the cache, a new version of StatStack is used to model write invalidation in the private cache, as well as positive and negative interference in the shared cache.

During the second phase, synchronization overhead is modeled to predict the total execution time of the application. Synchronization is first profiled by tracking function calls to the pthread and OpenMP libraries. Synchronization calls we model include barriers, critical sections, joins and condition variables, but we are convinced that our approach is general enough to work with even manually implemented synchronization methods.

An in-depth evaluation using 26 benchmarks from the Rodinia and Parsec benchmark suites, using both the OpenMP and pthread libraries, showed that RPPM predicts performance within 11.5% on average (and 23% max).

RPPM is useful when doing design space exploration and characterization studies of multi-threaded applications. We illustrate that RPPM can be used to prune a design space by selecting (near-)optimal designs. When selecting designs within a 5% margin of the best design, we find the best design for all but one application and only selected a second design point for 5 applications.

RPPM can also be used to gain insight into the synchronization behavior of multi-threaded applications. This is shown by constructing bottle graphs, which show how the threads in these multi-threaded applications behave. Two other case studies show that the model can provide insight into the difference between using the OpenMP and pthread libraries, as well as the impact of time sharing from threads on cores.

9.3 Future Work

In this section, we give an overview of the potential further improvements that could be made to the work described in this thesis.

9.3.1 Power Efficiency

Linear branch entropy is a useful metric for classifying branches. Hard-to-predict branches have a high likelihood of resulting in an incorrect branch outcome prediction. When this happens, the processor starts fetching, decoding and executing instructions on the incorrect path. Only when the processor executes the branch, it will correct this mistake and nullify instructions.

Instead, the processor could just stall the fetch stage and wait for the branch to get executed. Waiting means that the processor does not need to spend power and energy on these wrong-path instructions, but waiting could also result in a performance penalty. So only stalling on branches with a lot of incorrect predictions is very important. Therefore the possible energy reduction should be investigated and should be weighted against the potential performance loss.

9.3.2 Impact of Time Sharing and SMT

As shown in the case study using dedup and ferret in Section 8.3, the impact of time sharing can be significant. To accurately model time sharing, RPPM should be extended to model the behavior of a scheduler.

When the scheduler puts two threads on the same core, this implies that they would share the core as well as the private cache. An important consequence of this is that the separation between phase one and phase two of the RPPM prediction model is no longer possible. During phase one of the model, the per-epoch active execution time is calculated, later in the second phase the synchronization is modeled and the total execution time of the application is predicted. This execution time and synchronization behavior would be an important input to the scheduling model, but the scheduling proposed by this new model will have an impact on the calculations from phase one, and thus on phase two and thus on the scheduling, etc.

SMT or simultaneous multi-threading also schedules two (or more) threads on the same core, but instead of switching between threads every couple milliseconds, threads are executed simultaneously. So, all threads can be actively executing instructions at the same time, thus not only sharing memory, but sharing every stage of the processor pipeline. This means that further research is needed to adapt the core model to accurately model interference in the reorder buffer, issue buffers, contention of ALUs, etc.

9.3.3 Message Passing Interface (MPI)

RPPM supports both OpenMP and pthread libraries by profiling the synchronization behavior by catching functions calls to these libraries. The same approach could be used to model applications that use MPI as the way to synchronize their progress. The power of MPI is that it can be used to communicate to systems over the network in order to speed up applications beyond the capabilities of a single computer.

To accurately predict the execution time of applications communicating over the network, a network model should be added to RPPM. Research should be done to find out if a network model exists that is suitable to integrate into the synchronization model. This model would predict the communication time needed to pass messages between nodes over the network.

9.3.4 Heterogeneous Multi-threading

RPPM uses a new version of StatStack to model the memory behavior of multi-threaded applications. To model cache interference in the private and shared caches, it records memory operations during the profiling phase. These memory operations are ordered as they are executed during the profiling phase. This ordering is later used to model write invalidation in private caches and positive or negative interference in the shared cache. Therefore, accurately modeling heterogeneous multicore systems, where the ordering of the memory operations could be totally different, is not possible.

To accurately model these heterogeneous systems, a solution has to be developed to avoid the ordering of the memory operations during profiling. To predict write invalidation, without ordering the samples, we need to sample all accesses to the same address. One possible approach to do this, without an infeasable increase in the number of samples, could be to use a different sampling technique. StatStack currently uses random sampling. However, with a prerun we could possibly identify interesting memory operations, and sample these during the profiling phase. When modeling write invalidation, the memory accesses can be reordered as if they were executing on the configuration to model (not the configuration where the profiling was done).

Bibliography

[1] G. Åhlman. Microarchitecture-independent data locality analysis of multithreaded applications on multicore processors. Master's thesis, Uppsala University, Division of Computer Systems, 2016.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 177–189, January 1983.

[3] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] N. Binkert, B. Beckmann, G. Black, S. K Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. volume 39, pages 1–7, 2011.

[6] M. Breughe. Maximizing Branch Behavior Coverage for a Limited Simulation Budget. `http://http://www.jilp.org/cbp2016/slides/mbreughe_WorkloadSelection.pptx`, 2016. [Online; accessed 18 October 2018].

[7] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2011.

[8] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.*, 11(3):28:1–28:25, August 2014.

[9] P. Chang, E. Hao, T. Yeh, and Y. Patt. Branch classification: A new mechanism for improving branch predictor performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*, pages 22–31, 1994.

[10] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, October 2009.

[12] I. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 128–137, October 1996.

[13] X. E. Chen and T. M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 59–70, December 2008.

[14] Y. Choi, A. Knies, L. Gerke, and T. Ngai. The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 182–191, December 2001.

[15] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pages 468–477, Oct 1996.

[16] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 355–372, 2013.

[17] L. Eeckhout and L. K. John. *Performance evaluation and benchmarking.* CRC Press, 2005.

[18] D. Eklöv and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 55–65, March 2010.

[19] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Performance Counter Architecture for Computing Accurate CPI Components. In *Proceedings of The Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–184, October 2006.

[20] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):42–53, May 2009.

[21] S. Eyerman, J. E. Smith, and L. Eeckhout. Characterizing the branch misprediction penalty. In *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 48–58, March 2006.

[22] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–157, 1993.

[23] A. Hartstein and T. R. Puzak. The optimal pipeline depth for a microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 7–13, May 2002.

[24] M. Haungs, P. Sallee, and M. Farrens. Branch transition rate: A new metric for improved branch classification analysis. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 241–250, 2000.

[25] K.M. Hazelwood and T.M. Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 71–80, October 2000.

[26] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[27] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[28] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 195–206, October 2006.

[29] Y. Ishii. Global-local combined branch history: The alternative way to improve TAGE branch predictor. In *JWAC-4: Championship Branch Prediction*. JILP, June 2014.

[30] D. Jiménez. Strided sampling hashed perceptron predictor. In *JWAC-4: Championship Branch Prediction*. JILP, June 2014.

[31] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA)*, pages 197–206, Jan 2001.

[32] R. Jongerius, A. Anghel, G. Dittmann, G. Mariani, E. Vermij, and H. Corporaal. Analytic multi-core processor model for fast design-space exploration. *IEEE Transactions on Computers*, 67(6):755–770, June 2018.

[33] A. Joshi, A.and Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, 2006.

[34] T. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 338–349, June 2004.

[35] T. S. Karkhanis and J. E. Smith. Automated design of application specific superscalar processors: An analytical approach. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 402–411, 2007.

[36] P. M. Kogge. *The architecture of pipelined computers.* CRC press, 1981.

[37] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 185–194, October 2006.

[38] B. Lee, D. Brooks, Bronis R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 249–258, March 207.

[39] B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 270–281, November 2008.

[40] B. C. Lee and D. M. Brooks. Illustrative design space studies with microarchitectural regression models. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, February 2007.

[41] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 190–200, June 2005.

[42] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. Simics/sun4m: A virtual workstation. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, pages 10–10, 1998.

[43] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution

on branch prediction. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 217–227, December 1994.

[44] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 138–149, June 1995.

[45] S. McFarling. Combining branch predictors. Technical Report WRL TN-36, Digital Western Research Laboratory, June 1993.

[46] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 2–10, 1999.

[47] S. Otiv, K. Garikipati, M. Patnaik, and V. Kamakoti. H-pattern: A hybrid pattern based dynamic branch predictor with performance based adaptation. In *JWAC-4: Championship Branch Prediction*. JILP, June 2014.

[48] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham. Using model trees for computer architecture performance analysis of software applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 116–125, April 2007.

[49] M. Popov, C. Akel, F. Conti, W. Jalby, and P. d. O. Castro. PCERE: Fine-grained parallel benchmark decomposition for scalability prediction. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1151–1160, May 2015.

[50] A. Seznec. Analysis of the O-GEometric history length branch predictor. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 394–405, June 2005.

[51] A. Seznec. TAGE-SC-L branch predictors. In *JWAC-4: Championship Branch Prediction*. JILP, June 2014.

[52] Y. S. Shao and D. Brooks. ISA-independent workload characterization and its implications for specialized architectures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255, 2013.

[53] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, 2002.

[54] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, October 2002.

[55] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA)*, pages 135–148, 1981.

[56] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Micro-architecture independent analytical processor performance and power modeling. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 32–41, 2015.

[57] S. Van den Steen and L. Eeckhout. Modeling superscalar processor memory-level parallelism. *Computer Architecture Letters*, 1(2):10–13, June 2018.

[58] S. Van den Steen, S. Eyerman, S. De Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Analytical processor performance and power modeling using micro-architecture independent characteristics. *IEEE Transactions on Computers*, 65(12):3537–3551, 2016.

[59] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 84–95, June 2003.

[60] T. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th annual international symposium on computer architecture (ISCA '93)*, pages 257–266, 1993.

[61] T. Yokota, K. Ootsu, and T. Baba. Potentials of branch predictors: From entropy viewpoints. In *Proceedings of the 21st International Conference on Architecture of Computing Systems (ARCS)*, pages 273–285, 2008.

[62] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS)*, pages 23–34, 2007.